

UNISYS

CTOS®

Microsoft Macro

**Assembler Programming and
Installation Reference
Manual**

Release 5.1

Priced Item

June 1991
Printed in US America
4164 0574-000

UNISYS

CTOS®

Microsoft Macro

**Assembler Programming and
Installation Reference
Manual**

Copyright © 1991 Unisys Corporation
All rights reserved.

Unisys is a registered trademark of Unisys Corporation
CTOS is a registered trademark of Convergent Technologies, Inc.
a wholly owned subsidiary of Unisys Corporation.
Microsoft is a registered trademark of Microsoft Corporation.

Release 5.1

Priced Item

June 1991

Printed in US America

4164 0574-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a User Communication Form (UCF) with the CLASS specified as "Documentation", the Type specified as "Trouble Report", and the product specified as the title and part number of the manual (for example, 4164 0574-000).

BTOS, CTOS/VM, and Task Manager are trademarks of Unisys Corporation.

IBM, OS/2 and XENIX are registered trademarks of International Business Machines Corporation.

MS-DOS is a registered trademark and MS OS/2 is a trademark of Microsoft Corporation.

Intel is a registered trademark of Intel Corporation.

Page Status

Page	Issue
iii through vii	Original
viii	Blank
ix through xix	Original
xx	Blank
xxi through xxiv	Original
1-1 through 1-2	Original
2-1 through 2-3	Original
2-4	Blank
3-1 through 3-8	Original
4-1 through 4-29	Original
4-30	Blank
5-1 through 5-7	Original
5-8	Blank
6-1 through 6-16	Original
7-1 through 7-35	Original
7-36	Blank
8-1 through 8-24	Original
9-1 through 9-15	Original
9-16	Blank
10-1 through 10-12	Original
11-1 through 11-26	Original
12-1 through 12-11	Original
12-12	Blank
13-1 through 13-26	Original
14-1 through 14-9	Original
14-10	Blank
15-1 through 15-15	Original
15-16	Blank
16-1 through 16-16	Original
17-1 through 17-16	Original
18-1 through 18-26	Original
19-1 through 19-27	Original
19-28	Blank

Page	Issue
20-1 through 20-14	Original
21-1 through 21-34	Original
22-1 through 22-4	Original
A-1 through A-5	Original
A-6	Blank
B-1 through B-26	Original
Glossary-1 through 8	Original
Index-1 through 11	Original
Index-12	Blank

About This Manual

The CTOS Microsoft Macro Assembler allows you to write, assemble, link, and run assembly language programs. This reference manual contains information about installing the software, writing assembly language programs, and using the Cross-Reference Utility supplied with the CTOS Microsoft Macro Assembler package.

Who Should Use This Manual

This manual is for experienced assembly language programmers who are familiar with CTOS.

How to Use This Manual

If you are using the CTOS Microsoft Macro Assembler for the first time, you should read sections 1 through 4. They contain an overview of the assembler's features and installation instructions.

If you scan the table of contents and review the topics before you start, you will find this manual easier to use. To locate specific information, use the index.

How This Manual Is Arranged

The material in this manual is divided into 22 sections and two appendixes. It contains an index and a glossary.

For information on:

See:

The CTOS Microsoft Macro Assembler in general and its differences from the Microsoft Macro Assembler Version 5.1	Section 1
Installing the software	Section 2
Setting up your MASM environment	Section 3
Using MASM and its options	Section 4
Using the cross-reference listing utility	Section 5
Using assembly language statements to write source code	Section 6
Full and simplified segment structures and segment groups	Section 7
Defining labels and variables	Section 8
Using structures and records	Section 9
Creating programs from multiple modules	Section 10
Using operands and expressions	Section 11
Conditional assembly	Section 12
Using equates, macros, and repeat blocks	Section 13
Controlling assembly output	Section 14
The 8086 family processors including the 80386	Section 15
Addressing modes	Section 16
Loading, storing, and moving data	Section 17
Arithmetic and bit manipulation	Section 18
Controlling program flow	Section 19
Processing strings	Section 20
Using a math coprocessor	Section 21
Controlling the processor	Section 22
New features in Version 5.1.0	Appendix A
Error messages and exit codes	Appendix B

Conventions

The following conventions apply throughout this manual:

- When two keys are used together for an operation, their names are hyphenated (for example, ACTION-GO).
- Optional material is enclosed in double brackets ([]).
- Programmer supplied variables are shown in *italics*.
- When the programmer is to choose between two or more options, they are shown in braces separated by a vertical line (for example, {option1 | option2 | option3}).
- The term “character” includes the space.

Example Programs

Short example code sequences and occasional complete example programs appear throughout this manual.

Related Product Information

For an explanation of CTOS Executive commands and using the Executive command line, you can refer to the *CTOS Executive Reference Manual* and the *CTOS Executive Operations Guide*, respectively.

For information on running multiple application contexts under CTOS Context/Window Manager, see the *CTOS Context/Window Manager Installation and Configuration Guide, Volume 2: Protected Mode*.

Contents

About This Guide	v
 Section 1. Overview	
CTOS Microsoft Macro Assembler Components	1-1
Assembler	1-1
Cross-Reference Utility	1-2
Features New to Version 5.1.1	1-2
 Section 2. CTOS Microsoft Macro Assembler Installation	
Installing the CTOS Microsoft Macro Assembler	2-1
Using Software Installation	2-2
Using Install	2-3
 Section 3. Getting Started	
Choosing a Configuration Strategy	3-1
Setting Environment Variables	3-2
The Program Development Cycle	3-2
Developing Programs	3-5
Writing and Editing Assembly Language Source Code	3-5
Assembling Source Files	3-7
Converting Cross Reference Files	3-8
Creating Library Files	3-8
Linking Object Files	3-8
Debugging	3-8

Section 4. Using MASM

Running the Assembler	4-1
Assembly Using a Command Line	4-2
Assembly Using Prompts	4-4
Using Environment Variables	4-5
The INCLUDE Environment Variable	4-5
The MASM Environment Variable	4-6
Controlling Message Output	4-7
Using MASM Options	4-8
Specifying the Segment-Order Method	4-9
Setting the File-Buffer Size	4-10
Creating a Pass 1 Listing (/D)	4-10
Defining Assembler Symbols	4-11
Creating Code for a Floating Point Emulator (/E)	4-12
Getting Command Line Help (/H)	4-13
Setting a Search Path for Include Files (/I)	4-13
Specifying Listing and Cross Reference Files	4-14
Specifying Case Sensitivity	4-14
Suppressing Tables in the Listing File	4-15
Checking for Impure Code	4-15
Controlling Display of Assembly Statistics	4-16
Setting the Warning Level	4-16
Listing False Conditionals	4-18
Displaying Error Lines on the Screen	4-18
Writing Symbolic Information to the Object File	4-18
Reading Assembly Listings	4-19
Reading Code in a Listing	4-20
Reading a Macro Table	4-23
Reading a Structure and Record Table	4-24
Reading a Segment and Group Table	4-25
Reading a Symbol Table	4-26
Reading Assembly Statistics	4-28
Reading a Pass 1 Listing	4-28

Section 5. Using CREF

Using CREF	5-1
Using a Command Line to Create a Cross Reference Listing	5-2
Using Prompts to Create a Cross Reference Listing ..	5-3
Reading Cross Reference Listings	5-3

Section 6. Writing Source Code

Writing Assembly Language Statements	6-1
Using Mnemonics and Operands	6-3
Writing Comments	6-3
Assigning Names to Symbols	6-4
Constants	6-7
Integer Constants	6-7
Specifying Integers with Radix Specifiers	6-8
Setting the Default Radix	6-9
Packed Binary Coded Decimal Constants	6-9
Real Number Constants	6-10
String Constants	6-11
Defining Default Assembly Behavior	6-12
Ending a Source File	6-16

Section 7. Defining Segment Structure

Simplified Segment Definitions	7-2
Understanding Memory Models	7-3
Defining the Memory Model	7-4
Defining Simplified Segments	7-5
Using Predefined Equates	7-8
Simplified Segment Defaults	7-10
Default Segment Names	7-10
Full Segment Definitions	7-14
Setting the Segment Order Method	7-14
Defining Full Segments	7-16
Controlling Alignment with Align Type	7-17
Setting Segment Word Size with Use Type	7-17
Defining Segment Combinations with Combine Type	7-19
Controlling Segment Structure with Class Type ..	7-23
Defining Segment Groups	7-25
Associating Segments with Registers	7-28
Initializing Segment Registers	7-30
Initializing the CS and IP Registers	7-30
Initializing the DS Register	7-32
Initializing the SS and SP Registers	7-33
Initializing the ES Register	7-34
Nesting Segments	7-34

Section 8. Defining Labels and Variables

Using Type Specifiers	8-1
Defining Code Labels	8-3
Near Code Labels	8-3
Procedure Labels	8-4
Code Labels Defined with the LABEL Directive	8-5
Defining and Initializing Data	8-6
Variables	8-6
Integer Variables	8-7
Binary Coded Decimal Variables	8-11
String Variables	8-12
Pointer Variables	8-12
Real Number Variables	8-14
Initializing and Allocating Real Number Variables	8-14
Selecting a Real Number Format	8-15
Real Number Encoding	8-16
Arrays and Buffers	8-20
Labeling Variables	8-21
Setting the Location Counter	8-22
Aligning Data	8-23

Section 9. Using Structures and Records

Structures	9-1
Declaring Structure Types	9-2
Defining Structure Variables	9-3
Using Structure Operands	9-4
Records	9-5
Declaring Record Types	9-6
Defining Record Variables	9-8
Using Record Operands and Record Variables	9-11
Record Operators	9-12
The MASK Operator	9-12
The WIDTH Operator	9-13
Using Record Field Operands	9-14

Section 10. Creating Programs from Multiple Modules

Declaring Symbols Public	10-3
Declaring Symbols External	10-4
Using Multiple Modules	10-7
Declaring Symbols Communal	10-9

Section 11. Using Operands and Expressions

Using Operands with Directives	11-2
Using Operators	11-3
Calculation Operators	11-4
Arithmetic Operators	11-4
Structure Field Name Operator	11-6
Index Operator	11-6
Shift Operators	11-8
Bitwise Logical Operators	11-8
Relational Operators	11-9
Segment Override Operators	11-10
Type Operators	11-11
PTR Operator	11-12
SHORT Operator	11-12
THIS Operator	11-13
HIGH and LOW Operators	11-13
SEG Operator	11-14
OFFSET Operator	11-14
.TYPE Operator	11-15
TYPE Operator	11-16
LENGTH Operator	11-17
SIZE Operator	11-18
Operator Precedence	11-19
Using the Location Counter	11-21
Using Forward References	11-22
Forward References to Labels	11-22
Forward References to Variables	11-25
Strong Typing for Memory Operands	11-25

Section 12. Assembling Conditionally

Using Conditional Assembly Directives	12-1
Testing Expressions with IF and IFE Directives	12-3
Testing the Pass with IF1 and IF2 Directives	12-3
Testing Symbol Definition with IFDEF and IFNDEF	
Directives	12-4
Verifying Macro Parameters with IFB and IFNB	
Directives	12-4
Comparing Macro Arguments with IFIDN and IFDIF	
Directives	12-5

Using Conditional Error Directives	12-6
Generating Unconditional Errors with .ERR, .ERR1, and .ERR2 Directives	12-7
Testing Expressions with .ERRE or .ERRNZ Directives	12-8
Verifying Symbol Definitions with .ERRDEF and .ERRNDEF Directives	12-9
Testing for Macro Parameters with .ERRB and .ERRNB Directives	12-10
Comparing Macro Arguments with .ERRIDN and .ERRDIF Directives	12-10

Section 13. Using Equates, Macros, and Repeat Blocks

Using Equates	13-2
Redefinable Numeric Equates	13-2
Nonredefinable Numeric Equates	13-3
String Equates	13-4
Using Macros	13-5
Defining Macros	13-6
Calling Macros	13-8
Using Local Symbols	13-9
Exiting from a Macro	13-10
Defining Repeat Blocks	13-11
The REPT Directive	13-12
The IRP Directive	13-12
The IRPC Directive	13-13
Using Macro Operators	13-14
Substitute Operator	13-15
Literal Text Operator	13-16
Literal Character Operator	13-17
Expression Operator	13-18
Macro Comments	13-19
Using Recursive, Nested, and Redefined Macros	13-19
Using Recursion	13-20
Nesting Macro Definitions	13-21
Nesting Macro Calls	13-22
Redefining Macros	13-23
Avoiding Inadvertent Substitutions	13-23
Managing Macros and Equates	13-24
Using Include Files	13-24
Purging Macros from Memory	13-25

Section 14. Controlling Assembly Output

Sending Messages to the Standard Output Device	14-1
Controlling Page Format in Listings	14-2
Setting the Listing Title	14-2
Setting the Listing Subtitle	14-3
Controlling Page Breaks	14-3
Controlling the Contents of Listings	14-5
Suppressing and Restoring Listing Output	14-5
Controlling Listing of Conditional Blocks	14-6
Controlling Listing of Macros	14-7
Controlling Cross Reference Output	14-8

Section 15. Understanding 8086 Family Processors

Using the 8086 Family Processors	15-1
Processor Differences	15-2
Real and Protected Modes	15-3
Segmented Addresses	15-4
Using 8086 Family Registers	15-6
Segment Registers	15-9
General Purpose Registers	15-9
Other Registers	15-11
The Flags Register	15-12
8087 Family Registers	15-14
Using the 80386 Processor	15-15

Section 16. Using Addressing Modes

Using Immediate Operands	16-2
Using Register Operands	16-3
Using Memory Operands	16-4
Direct Memory Operands	16-4
Indirect Memory Operands	16-6
80386 Indirect Memory Operands	16-11

Section 17. Loading, Storing, and Moving Data

Transferring Data	17-1
Copying Data	17-2
Exchanging Data	17-3
Looking Up Data	17-3
Transferring Flags	17-4

Converting between Data Sizes	17-5
Extending Signed Values	17-5
Extending Unsigned Values	17-6
Moving and Extending Values	17-6
Loading Pointers	17-7
Loading Near Pointers	17-8
Loading Far Pointers	17-9
Transferring Data to and from the Stack	17-10
Pushing and Popping	17-10
Using the Stack	17-13
Saving Flags on the Stack	17-14
Saving All Registers on the Stack	17-15
Transferring Data to and from Ports	17-15

Section 18. Doing Arithmetic and Bit Manipulations

Adding	18-1
Adding Values Directly	18-1
Adding Values in Multiple Registers	18-3
Subtracting	18-4
Subtracting Values Directly	18-4
Subtracting with Values in Multiple Registers	18-6
Multiplying	18-7
Dividing	18-9
Calculating with Binary Coded Decimals	18-11
Unpacked BCD Numbers	18-12
Packed BCD Numbers	18-14
Doing Logical Bit Manipulations	18-15
AND Operations	18-16
OR Operations	18-17
XOR Operations	18-17
NOT Operations	18-18
Scanning for Set Bits	18-19
Shifting and Rotating Bits	18-20
Multiplying and Dividing by Constants	18-22
Moving Bits to the Least Significant Position	18-24
Adjusting Masks	18-24
Shifting Multiword Values	18-25
Shifting Multiple Bits	18-25

Section 19. Controlling Program Flow

Jumping	19-1
Jumping Unconditionally	19-2
Jumping Conditionally	19-4
Comparing and Jumping	19-4
Jumping Based on Flag Status	19-8
Testing Bits and Jumping	19-9
Testing and Setting Bits	19-10
Looping	19-12
Setting Bytes Conditionally	19-14
Using Procedures	19-15
Calling Procedures	19-16
Defining Procedures	19-17
Passing Arguments on the Stack	19-19
Using Local Variables	19-21
Setting Up Stack Frames	19-24
Using Interrupts	19-25
Calling Interrupts	19-25

Section 20. Processing Strings

Setting Up String Operations	20-2
Moving Strings	20-5
Searching Strings	20-7
Comparing Strings	20-9
Filling Strings	20-10
Loading Values from Strings	20-11
Transferring Strings to and from Ports	20-13

Section 21. Calculating with a Math Coprocessor

Coprocessor Architecture	21-2
Coprocessor Data Registers	21-2
Coprocessor Control Registers	21-3
Emulation	21-5
Using Coprocessor Instructions	21-5
Using Implied Operands in the Classical Stack Form ..	21-7
Using Memory Operands	21-8
Specifying Operands in the Register Form	21-9
Specifying Operands in the Register Pop Form	21-10

Coordinating Memory Access	21-11
Transferring Data	21-12
Transferring Data to and from Registers	21-13
Real Transfers	21-14
Integer Transfers	21-15
Packed BCD Transfers	21-15
Loading Constants	21-17
Transferring Control Data	21-18
Doing Arithmetic Calculations	21-19
Addition	21-20
Normal Subtraction	21-20
Reversed Subtraction	21-21
Multiplication	21-22
Normal Division	21-22
Reversed Division	21-23
Other Operations	21-24
Controlling Program Flow	21-26
Comparing Operands to Control Program Flow	21-27
Compare	21-28
Compare and Pop	21-29
Testing Control Flags after Other Instructions	21-30
Using Transcendental Instructions	21-31
Controlling the Coprocessor	21-33

Section 22. Controlling the Processor

Controlling Timing and Alignment	22-1
Controlling the Processor	22-2
Controlling Protected Mode Processes	22-3
Controlling the 80386	22-4

Appendix A. New Features

MASM Enhancements	A-1
80386 Support	A-1
Segment Simplification	A-3
Performance Improvements	A-3
Enhanced Error Handling	A-3
New Options	A-4
Environment Variables	A-4
String Equates	A-4
RETF and RETN Instructions	A-5
Communal Variables	A-5
Flexible Structure Definitions	A-5

Appendix B. Error Messages and Exit Codes

MASM Messages and Exit Codes	B-1
Assembler Status Messages	B-1
Numbered Assembler Messages	B-2
Unnumbered Error Messages	B-23
File-Access Errors	B-23
Command-Line Errors	B-23
Miscellaneous Errors	B-24
MASM Exit Codes	B-25
CREF Error Messages and Exit Codes	B-26
 Glossary	 Glossary-1
 Index	 Index-1

Figures

3-1.	The Program Development Cycle	3-3
4-1.	CTOS Microsoft Macro Assembler Listing	4-22
5-1.	Example Assembly Listing	5-4
5-2.	Example Cross Reference Listing	5-7
7-1.	Segment Structure with Combine and Align Types	7-22
7-2.	Segment Structure with Groups	7-27
8-1.	Define Word Directive	8-9
8-2.	Define Doubleword Directive	8-10
8-3.	Define Quadword Directive	8-10
8-4.	Far Pointer	8-13
8-5.	Far Pointer in 32 Bit Mode	8-13
8-6.	Encoding for Real Numbers in IEEE Format	8-17
8-7.	Encoding for Real Numbers in Microsoft Binary Format	8-18
8-8.	Encoding for Real Numbers in Temporary Real Format	8-19
9-1.	Byte record	9-7
9-2.	16 Bit Record	9-8
9-3.	Byte Record	9-10
9-4.	Type Declaration	9-10
9-5.	Record Operand Values	9-12
15-1.	Register for 8088-80286 Processors	15-7
15-2.	Extended Registers of 80386 Processor	15-8
15-3.	Flags for 8088-80386 Processors	15-13
17-1.	Stack Status after Pushes and Pops	17-12
18-1.	Shifts and Rotates	18-21
19-1.	Procedure Arguments on the Stack	19-20
19-2.	Local Variables on the Stack	19-23
19-3.	Operation of Interrupts	19-27

Figures

21-1.	Coprocessor Data Registers	21-3
21-2.	Coprocessor Control Register	21-4
21-3.	Status of Register Stack	21-7
21-4.	Status of Register Stack and Memory Locations	21-9
21-5.	Register Stack with fxch Instruction	21-10
21-6.	Register Stack with faddp Instruction	21-11
21-7.	Register Stack Using Packed BCD	21-16
21-8.	Coprocessor and Processor Control Flags	21-27

Tables

3-1.	Program Development Environment Variables	3-1
4-1.	MASM Options	4-8
4-2.	Warning Levels	4-17
4-3.	Symbols and Abbreviations in Listings	4-21
5-1.	Example Assembly Listings	5-4
5-2.	Example Cross Reference Listing	5-7
6-2.	Digits Used with Each Radix	6-8
7-1.	Memory Models	7-3
7-2.	Predefined Equates	7-8
7-3.	Default Segments and Types for Standard Memory Models	7-11
7-4.	Align Types	7-17
7-5.	Combine Types	7-19
8-1.	Type Specifiers For Memory Operand Sizes	8-2
8-2.	Type Specifiers For Constants	8-2
8-3.	Assembly Directives	8-7
8-4.	Directive Descriptions	8-8
11-1.	Arithmetic Operators	11-4
11-2.	Logical Operators	11-8
11-3.	Relational Operators	11-10
11-4.	.TYPE Operator and Variable Attributes	11-16
11-5.	Operator Precedence	11-20
12-1.	Conditional Error Directives	12-7
13-1.	Macro Operators	13-14
15-1.	Segment Registerst	15-9
15-2.	General Purpose Register	15-10
15-3.	Flags	15-13
16-1.	Register Operands	16-3
16-2.	Indirect Addressing Modes	16-7
18-1.	Values Returned by Logical Operations	18-15

Tables

19-1.	Conditional-Jump Instructions Used after Compare	19-6
20-1.	Requirements for String Instructions	20-4
21-1.	Coprocessor Operand Forms	21-6
21-2.	Control Flag Settings after Compare or Test	21-28
A-1.	80386 and 80387 Instruction	A-2

Section 1

Overview

The CTOS Microsoft Macro Assembler is ported from the Microsoft Macro Assembler Version 5.1. It allows you to write, assemble, link, and run programs across the CTOS, DOS, and OS/2 platforms.

The CTOS Microsoft Macro Assembler runs on the following operating systems:

- BTOS II 3.2.0 or 3.0.2
- CTOS II 3.3.0
- CTOS/XE 3.0.0 (includes BTOS II 3.2.0)

This overview briefly describes the components of the CTOS Microsoft Macro Assembler package and lists the features new to Version 5.1.1 of the Microsoft Macro Assembler.

CTOS Microsoft Macro Assembler Components

The CTOS Microsoft Macro Assembler package contains the following components:

- MASM, the assembler program
- CREF, the cross-reference utility

The following text briefly describes these components.

Assembler

The CTOS Microsoft Macro Assembler allows you to assemble one or more assembly language source files. It creates object modules which are linked together to form run files. These object modules conform to the standard CTOS object module format and can be linked with object modules created with high level languages.

Cross-Reference Utility

The Cross-Reference Utility allows you to create cross references listings from binary cross-reference files produced by the assembler. The cross-reference listing are then used as a debugging aid when used in conjunction with the symbol table generated by the assembler.

Features New to Version 5.1.1

If you have used an earlier version of Microsoft MASM, you will find that Version 5.1.1 contains many new capabilities and allows you to perform familiar operations more quickly. MASM has been improved to support:

- The 80386 processor.
- Segment simplification which allows easier definition of segments.
- Faster assembly and larger symbol space.
- Enhanced error handling.
- New command line options.
- Environment variables.
- String equates.
- RETF and RETN instructions.
- Communal variables.
- Flexible structure definitions.

Section 2

CTOS Microsoft Macro Assembler Installation

This section contains migration and installation information.

Installing the CTOS Microsoft Macro Assembler

The following text provides procedures to install the CTOS Microsoft Macro Assembler 5.1.1 on your workstation using the following commands:

- Software Installation

Requires BTOS II 3.0 or higher, CTOS/VM 2.3 or higher, CTOS II, CTOS III, or CTOS/XE 3.0, with Standard Software level 12.0.

- Install

Requires BTOS II 3.0 or higher, CTOS/VM 2.3 or higher, CTOS II, CTOS III, or CTOS/XE 3.0, with Standard Software level 12.0.

Refer to the Software Release Announcement included in your CTOS Microsoft Macro Assembler package for details on system requirements.

Refer to the README.DOC file included with the software for information on installing and using the Task Manager.

Using Software Installation

To install CTOS Microsoft Macro Assembler using the Software Installation command, use the following procedure:

1. Load disk B25MSA-1 into a floppy drive.
2. Type **Software Installation** at the Executive command line.
3. Press RETURN.

The Software Installation command form appears:

Software Installation

[Cmd file]

[Files to]

[Install file]

4. Enter information into the command fields or accept the defaults, as described in the following table.

Optional Field	Entry
[Cmd file]	The default is the standard system command file, [Sys] <Sys> Sys.cmds. To specify a different command file, enter the file name or complete file specification.
[Files to]	The default is [Sys] <Sys> . To specify a different directory, enter the directory name.
[Install file]	The default installation file is [f0] <Sys> Install.sub. If the application program uses a different installation file, specify the file name in this field.

5. Press GO.

Software Installation dialog screens will guide you through the remainder of the installation procedure.

If you make a mistake, press ACTION-FINISH to cancel installation and begin again.

Using Install

To install the CTOS Microsoft Macro Assembler using the Install command, use the following procedure:

1. Load disk B25MSA-1 into a floppy drive.
2. Type **Install** at the Executive command line.
3. Press RETURN.

The Install command form appears:

Install [Floppy drive ([f0])]

4. Enter a drive specification into the command field or accept the default, as described in the following table.

Optional Field	Entry
[Floppy drive ([f0])]	The default drive is [f0]. To install software from a different drive, enter its device name.

5. Press GO.

Install dialog screens will guide you through the remainder of the installation procedure.

If you make a mistake, press **ACTION-FINISH** to cancel installation and begin again.

Section 3

Getting Started

This section tells you how to set up CTOS Microsoft Macro Assembler files and to start writing assembly language programs. It gives an overview of the development process and shows examples using simple programs. It also refers you to the sections where you can learn more about each subject.

Choosing a Configuration Strategy

Program development can be affected by the environment variables described below:

Table 3-1. Program Development Environment Variables

Variable	Description
PATH	Specifies the directories where DOS looks for executable files. A common setup with language products is to place executable files in the directory <BIN> and include this directory in the PATH environment string.
LIB	Specifies the directory where LINK looks for library and object files. A common setup with language products is to put library and object files in the directory <LIB> and include this directory in the LIB environment string.
INCLUDE	Specifies the directory where MASM looks for include files. A common setup with language products is to put macro files and other include files in the directory <INCLUDE> and to put this directory in the INCLUDE environment string.
MASM	Specifies default options that MASM uses on start up.
LINK	Specifies default options that LINK uses on start up.

continued

Table 3–1. Program Development Environment Variables (cont.)

Variable	Description
TMP	Specifies the directory where LINK places temporary files if it needs to create them.
INIT	<p>Specifies the directory where MAKE looks for the file TOOLS.INI, which may contain inference rules.</p> <p>You will probably want to use environment variables to specify locations for library, macro, and executable files.</p> <p>If you already have other language products on a hard disk, you should consider how your assembler setup interacts with your other languages.</p> <p>Some users may prefer to have separate directories for library and include files for each language. Others may prefer to have all library and include files in the same directories. If you want all language files in the same directories, make sure you do not have any files with the same names as the ones provided with the CTOS Microsoft Macro Assembler.</p>

Setting Environment Variables

If you wish to use environment variables to establish default file locations and options, you will probably want to set the environment variables in your SYSINIT.JCL file. The setup program does not attempt to set any environment variables, so you must modify any batch files yourself.

The following lines could be added:

```
SET LIB=[SYS] <LIB>  
SET INCLUDE=[SYS] <INCLUDE>
```

Note: *The LIB environment variable is provided for compatibility with other operating systems and has no effect in CTOS.*

The Program Development Cycle

The program development cycle for assembly language is illustrated in Figure 3–1.

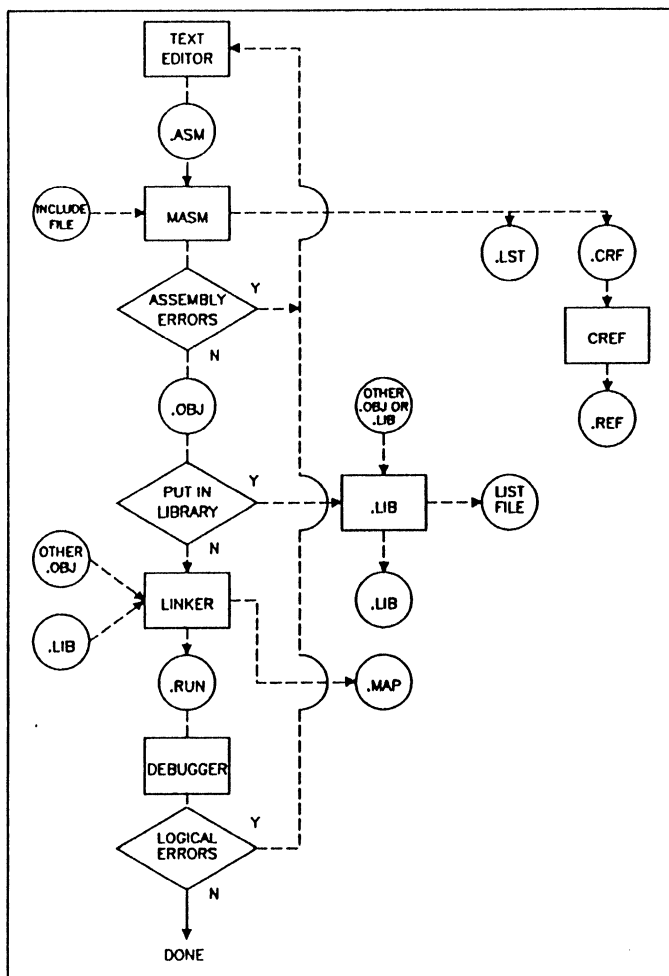


Figure 3-1. The Program Development Cycle

The specific steps for developing a stand alone assembler program are listed below:

- Use a text editor to create or modify assembly language source modules. By convention, source modules are given the extension `.ASM`. Source modules can be organized in a variety of ways. For instance, you can put all the procedures for a program into one large module, or you can split the procedures between modules. If your program will be linked with high level language modules, the source code for these modules is also prepared at this point.
- Use MASM to assemble each of the modules for the program. MASM may optionally read in code from include files during assembly. If assembly errors are encountered in a module, you must go back to Step 1 and correct the errors before continuing. For each source (`.ASM`) file, MASM creates an object file with the default extension `.OBJ`. Optional listing (`.LST`) and cross reference (`.CRF`) files can also be created during assembly. If your program will be linked with high level language modules, the source modules are compiled to object files at this point.
- Optionally use the CTOS Librarian to gather multiple object files (`.OBJ`) into a single library file having the default extension `.LIB`. It is generally used for object files that will be linked with several different programs. An optional library list file can also be created with the CTOS Librarian.
- Use the Linker to combine all the object files and library modules that make up a program into a single executable file with the extension `.RUN`. An optional `.MAP` file can also be created.
- Debug your program to discover logical errors. Debugging may involve several techniques, including the following:
 - Running the program and studying its input and output
 - Studying source and listing files
 - Using CREF to create a cross reference listing (`.REF`) file
 - Using an interactive debugger to locate and correct programming errors

If logical errors are discovered, you must return to Step 1 to correct the source code.

Developing Programs

The following text takes you through the steps involved in developing programs. Examples are shown for each step. The sections and manuals that describe each topic in detail are cross referenced.

Writing and Editing Assembly Language Source Code

Assembly language programs are created from one or more source files. Source files are text files that contain statements defining the program's data and instructions.

To create assembly language source files, you need a text editor capable of producing ASCII (American Standard Code for Information Interchange) files. Lines must be separated by a carriage return/line feed combination. If your text editor has a programming or nondocument mode for producing ASCII files, use that mode.

The following examples illustrate source code that produces a stand alone executable program. If you are a beginner to assembly language, you can start experimenting by copying these programs. Use the segment shell of the programs, but insert your own data and code.


```
; Declare the OS and object module procedures as external,  
; accessible by FAR CALLS  
  
    EXTRN WriteBsRecord: FAR, ErrorExit: FAR  
  
        TITLE HELLO  
        .MODEL small                ; Use small model  
  
        .DATA  
message    DB      "Hello, world.",13,10    ; Message to be written  
lmessage    EQU     $ - message            ; Length of message  
  
; We write to video using SAM's pre-opened bytestream  
; which is located in the data segment. It is important  
; to locate this declaration within the DATA SEGMENT as  
; shown here.  
  
    EXTRN  bsVid:  BYTE  
cbWrittenRet  DW      ?  
  
        .286                        ; allow protected mode opcodes  
        .CODE  
Start:  
; Here we will print the message using CTOS call WriteBsRecord  
  
        push    ds                    ; 1st arg is pbsVid  
        push    OFFSET bsVid  
        push    ds                    ; 2nd arg is prgcsMsg  
        push    OFFSET message  
        push    lmessage              ; 3rd arg is cbMsg  
        push    ds                    ; 4th arg is pcbWrittenRet  
  
        push    OFFSET cbWrittenRet  
        call    WriteBsRecord        ; make the call  
  
        push    ax                    ; AX contains "rcode"  
        call    ErrorExit  
  
        END      Start
```

Note the following points about the source file:

- The `.MODEL` directive tells MASM that you intend to use the small model. You then place segments in your source file in whatever order you find convenient using the `.STACK`, `.DATA`, `.CODE`, and other segment directives. You can still define the segments completely by using the directives required by earlier versions of MASM. The simplified segment directives and the Microsoft naming conventions are explained in Section 9.
- The `.DATA` directive marks the start of the data segment. A string variable and its length are defined in this segment.

- The `.286` directive allows you to include protected mode instructions. Without this directive, protected mode instructions are flagged as errors.
- The instruction label start in the code segment follows the `.CODE` directive and marks the start of the program instructions. The same label is used after the `END` statement to define the point where program execution will start. See Sections 8 and 9 for more information on using the `END` statement and defining the execution starting point.
- The string variable defined earlier is displayed using the `CTOS` call `WriteBsRecord`.
- `ErrorExit` is used to terminate the program. Note that `WriteBsRecord` returns an error code in the `ax` register. This value is passed on to `ErrorExit` upon termination. The executive will display any error code other than zero (normal termination).

Assembling Source Files

Source modules are assembled with MASM. The MASM command line syntax is:

MASM

```
[args] [[options] sourcefile [, [objectfile] [, [listingfile] [, [crossreferencefile]]]] [;]
```

Assume you had an assembly source file called `hello.asm`. For the fastest possible assembly, you could start MASM with the following command line:

MASM

```
[args] hello;
```

The output would be an object file called `hello.obj`.

The `/V` and `/Z` options instruct MASM to send additional statistics and error information to the screen during assembly. The output of the following command is three files: the object file `hello.obj`, the listing file `hello.lst`, and the cross reference file `hello.crf`.

MASM

```
[args] /V /Z hello,,;
```

Section 6 describes the MASM command line, options, and listing format in more detail.

Converting Cross Reference Files

Cross reference files produced by MASM are in a binary format and must be converted using CREF. The command line syntax is shown below:

CREF

[args] *crossreferencefile*, [[*crossreferencelisting*] [;]]

To convert the cross reference file *hello.crf* into an ASCII file that cross references symbols that are used in *hello.asm*, use the following command line:

CREF

[args] *hello*;

The output file is called *hello.ref*.

The CREF command line and listing format are described in Section 7.

Creating Library Files

Object files created with MASM or with high level language compilers can be converted to library files by using the CTOS Librarian.

For more information on the Librarian, refer to the *BTOS II Development Utilities Programming Reference Manual*.

Linking Object Files

Object files are linked into executable files using the Linker. For more information on the Linker refer to the *BTOS II Development Utilities Programming Reference Manual*.

Debugging

The CTOS debugger is available for debugging assembler programs. For more information on the CTOS Debugger, refer to the *CTOS Debugger User's Guide*.

Section 4

Using MASM

The CTOS Microsoft Macro Assembler (MASM) assembles 8086, 80186, 80286, and 80386 assembly language source files and creates relocatable object files. Object files can then be linked to form an executable file.

This section covers how to run MASM and explains the options and environment variables that control its behavior. It also describes the format of the assembly listings MASM generates. It contains the following subsections:

- Running the Assembler
- Using Environment Variables
- Controlling Message Output
- Using MASM Options
- Reading Assembly Listings

Running the Assembler

You can assemble source files with MASM by using two different methods: by entering a command at the executive or by responding to a series of prompts.

Once you have started MASM, it attempts to process the source file you specified. If errors are encountered, they are output to the screen and MASM terminates. If no errors are encountered, MASM outputs an object file. It can also output listing and cross reference files if they are specified. You can terminate MASM at any time by pressing ACTION-FINISH.

Assembly Using a Command Line

You can assemble a program source file by typing the MASM command name and entering the names of the files you wish to process on the first parameter line. The parameter line has the following form:

MASM

```
[args] [ options ] sourcefile [, [objectfile] [, [listingfile] [,
      [ crossreferencefile ]]]] [;]
```

The *options* can be any combination of the assembler options described in Using MASM Options. The option letter or letters must be preceded by a forward slash (/) or a dash (-). Examples in this manual use a forward slash. The forward slash and dash characters cannot be mixed in the same command line. Although shown at the beginning of the syntax line above, options may actually be placed anywhere on the command line. An option affects all relevant files in the command line even if the option appears at the end of the line.

The *sourcefile* must be the name of the source file to be assembled. If you do not supply a file name extension, MASM supplies the extension .ASM.

The optional *objectfile* is the name of the file to receive the relocatable object code. If you do not supply a name, MASM uses the source file name, but replaces the extension with .OBJ.

The optional *listingfile* is the name of the file to receive the assembly listing. The assembly listing shows the assembled code for each source statement and for the names and types of symbols defined in the program. If you do not supply a file name extension, the Macro Assembler supplies the extension .LST.

The optional *crossreferencefile* is the name of the file to receive the cross reference output. The resulting cross reference file can be processed with CREF, the Microsoft Cross Reference Utility, to create a cross reference listing of the symbols in the program. The cross reference listing can be used for program debugging. If you do not supply a file name extension, MASM supplies .CRF by default.

You can use a semicolon (;) anywhere after the *sourcefile* to select defaults for the remaining file names. A semicolon after the source file name selects a default object file name and suppresses creation of the assembly listing and cross reference files. A semicolon after the object file name suppresses just the listing and cross reference files. A semicolon after the listing file name suppresses only the cross reference file.

All files created during the assembly are written to the current drive and directory unless you specify a different drive for each file. You must separately specify the alternate drive and path for each file that you do not want to go on the current directory.

You can also specify a device name instead of a file name for example, NUL for no file, PRN for the default printer, or *Printrname* to select a different printer.

Note: *If you want the file name for a given file to be the default (the file name of the source file), place the commas that would otherwise separate the file name from the other names side by side (,,). Unless a semicolon (;) is used, all the commas in the command line are required.*

*Spaces in a command line are optional. If you make an error entering any of the file names, MASM displays an error message and prompts for new file names, using the method described in *Assembly Using Prompts*.*

Enter MASM with the following parameters:

MASM

[args] file.asm, file.obj, file.lst, file.crf

The example above is equivalent to the parameter line below:

MASM

[args] file,,;

The source file file.asm is assembled. The generated relocatable code is copied to the object file file.obj. MASM also creates an assembly listing and a cross reference file. These are written to file.lst and file.crf, respectively.

The following example directs MASM to assemble the source file `startup.asm`. The assembler then writes the relocatable object code to the default object file, `startup.obj`. MASM creates a listing file named `stest.lst`, but the semicolon keeps the assembler from creating a cross reference file.

MASM

```
[args] startup,,stest;
```

The following example is the same as the previous example except that the semicolon follows a comma that marks the place of the cross reference file. The assembler creates a cross reference file `startup.crf`.

MASM

```
[args] startup,,stest,;
```

The following example directs MASM to find and assemble the source file `build.asm` in the directory `<esrc>` on drive `[D1]`. The semicolon causes the assembler to create an object file named `build.obj` in the current directory, but prevents MASM from creating an assembly listing or cross reference file. Note that the object file is placed on the current drive, not the drive specified for the source file.

MASM

```
[args] [D1] <esrc> build;
```

Assembly Using Prompts

You can direct MASM to prompt you for the files it needs by starting MASM with just the command name. MASM prompts you for the input it needs by displaying the following lines, one at a time:

Source filename	[.ASM]:
Object filename	[source.OBJ]:
Source listing	[NUL.LST]:
Cross-reference	[NUL.CRF]:

The prompts correspond to the fields of MASM command lines. MASM waits for you to respond to each prompt before printing the next one. You must type a source file name (though the extension is optional) at the first prompt. For other prompts, you can either type a file name, or press the RETURN key to accept the default displayed in brackets after the prompt.

File names typed at prompts must follow the command line rules described in *Assembly Using a Command Line*. You can type options after any of the prompts as long as you separate them from file names with spaces. At any prompt, you can type the rest of the file names in the command line format. For example, you can choose the default responses for all remaining prompts by typing a semicolon (;) after any prompt (as long as you have supplied a source file name), or you can type commas (,) to indicate several files.

After you have answered the last prompt and pressed the RETURN key, MASM assembles the source file.

Using Environment Variables

The Macro Assembler recognizes two environment variables: INCLUDE and MASM. The subsections below describe these environment variables and their use with the assembler.

Environment variables are described in general in the DOS user's guide.

The INCLUDE Environment Variable

The INCLUDE environment variable specifies the directory where include files are stored. This makes maintenance of include files easier. All include files can be kept in the same directory. If you keep source files in different directories, you do not have to keep copies of include files in each directory.

The INCLUDE environment variable is used by MASM only if you give a file name as an argument to the INCLUDE directive (see Section 13). If you give a complete file specification, including directory or volume, MASM only looks for the file in the specified directory.

When a file name is specified, MASM looks for the include file first in any directory specified with the /I option (see *Setting a Search Path for Include Files*). If the /I option is not used or if the file is not found, MASM next looks in the current directory. If the file is still not found, MASM looks in the directories specified with the INCLUDE environment variable in the order specified.

SET INCLUDE=[SYS]<MSCInclude>

This line defines the INCLUDE environment string to be [SYS] <MSCInclude> . Include files placed in this directory can be found automatically by MASM. You can put this line in your file to set the environment string each time you turn on your workstation.

The MASM Environment Variable

The MASM environment variable can be used to specify default assembler options. If you define the options you use most in the environment variable, you do not need to type them on the command line every time you start the Macro Assembler.

When you start the assembler, it reads the options in the environment variable first. Then it reads the options in the command line. If conflicting options are encountered, the last one read takes effect. This means that you can override default options in the environment variable by giving conflicting options in the command line.

Some options define the default action. If given by themselves, they have no effect since the default action is taken anyway. However, they are useful for overriding a nondefault action specified by an option in the environment variable.

Some assembler directives have the same effect as options. They always override related options.

Note: *The equal sign (=) is not allowed in environment variables. Therefore, the /D option, when used with the equal sign, cannot be put in an environment variable. For example, the following command line is illegal and causes a syntax error:*

```
SET MASM=/Dtest=50
```

The following command line sets the MASM environment variable so that the /A, /ZI, and /Z options are in effect. The line can be put in your SYSINIT.JCL file to automatically set these options each time you start your workstation.

```
SET MASM= /A /ZI /Z
```

Assume you have set the MASM environment string using the line shown above, and you then start MASM with the following command line:

```
MASM /S test;
```

The /S option, which specifies sequential segment ordering, conflicts with the /A option, which specifies alphabetical segment ordering. The command line option overrides the environment option, and the source file has sequential ordering. (See Section 7 for information on the significance of segment order.) However, if the source file contains the .ALPHA directive, it overrides all options and specifies alphabetical segment order.

Controlling Message Output

During and immediately after assembly, MASM sends messages to the standard output device. By default, this device is the screen. However, the display can be redirected so that instead it goes to a file or to a device such as a printer.

The messages can include a status message for successful assembly and error messages for unsuccessful assembly. The message format and the error and warning messages are described in Appendix B.

Some text editing programs can use error information to locate errors in the source file. Typically, MASM is run as a shell from the editor and the assembler output is redirected into a file. The editor then opens the file and uses the data in it to locate errors in the source code. The errors may be located by line number, or by a search for the text of the error line.

If your text editor does not support this capability directly, you may still be able to use keystroke macros to set up similar functions. This requires an editor that supports keystroke macros.

MASM

```
[args] file; > errors
```

This command line sends to the file errors all messages that would normally be sent to the screen.

Using MASM Options

The MASM options control the operation of the assembler and the format of the output files it generates. Options can be entered with any combination of uppercase and lowercase letters.

MASM has the following options:

Table 4–1. MASM Options

Option	Action
<code>/A</code>	Writes segments in alphabetical order
<code>/Bnumber</code>	Sets buffer size (Not supported in CTOS)
<code>/C</code>	Specifies a cross reference file
<code>/D</code>	Creates Pass 1 listing
<code>/Dsymbol[=value]</code>	Defines assembler symbol
<code>/E</code>	Creates code for emulated floating point instructions
<code>/H</code>	Lists command line syntax and all assembler options
<code>/lpath</code>	Sets include file search path
<code>/L</code>	Specifies an assembly listing file
<code>/ML</code>	Makes names case sensitive
<code>/MU</code>	Converts names to uppercase letters
<code>/MX</code>	Makes public and external names case sensitive
<code>/N</code>	Suppresses tables in listing file
<code>/P</code>	Checks for impure code
<code>/S</code>	Writes segments in source code order
<code>/T</code>	Suppresses messages for successful assembly
<code>/V</code>	Displays extra statistics to screen
<code>/W{0 1 2}</code>	Sets error display level

continued

Table 4–1. MASM Options (cont.)

Option	Action
/X	Includes false conditionals in listings
/Z	Displays error lines on screen
/ZD	Puts line number information in the object file
/ZI	Puts symbolic and line number information in the object file

Note: *Previous versions of the assembler provided a /R option to enable 8087 instructions and real numbers in the IEEE format. Since the current version of the assembler enables 8087 instructions and IEEE format by default, the /R option is no longer needed. The option is still recognized so that old make and batch files will work, but it has no effect. The previous default format, Microsoft Binary, can be specified with the .MSFLOAT directive, as described in Section 6.*

Specifying the Segment–Order Method

The /A option directs MASM to place the assembled segments in alphabetical order before copying them to the object file. The default /S option directs the assembler to write segments in the order in which they appear in the source code.

Source code order is the default. If no option is given, MASM copies the segments in the order encountered in the source file. The /S option is provided for compatibility with the XENIX® operating system and for overriding a default option in the MASM environment variable.

Note: *Some previous versions of the IBM® Macro Assembler ordered segments alphabetically by default. Listings in some books and magazines have been written with these early versions in mind. If you have trouble assembling and linking a listing taken from a book or magazine, try using the /A option.*

The order in which segments are written to the object file is only one factor in determining the order in which they will appear in the executable file. The significance of segment order and ways to control it are discussed in Section 7.

The following command creates an object file, file.obj, whose segments are arranged in alphabetical order.

MASM

[args] /A file;

If the /S option were used instead, or if no option were specified, the segments would be arranged in sequential order.

Setting the File-Buffer Size

/B number (This option is not supported in CTOS.) The /B option directs the assembler to change the size of the file buffer used for the source file. The number is the number of 1024 byte (1K) memory blocks allocated for the buffer. You can set the buffer to any size from 1K to 63K. The default size of the buffer is 32K.

A buffer larger than your source file allows you to do the entire assembly in memory, greatly increasing assembly speed. However, you may not be able to use a large buffer if your workstation does not have enough memory or if you have too many resident programs using up memory. If you get an error message indicating insufficient memory, decrease the buffer size and try again.

You can enter MASM with different options with the following results:

/B16 file;

decreases the buffer size to 16K.

/B63 file;

increases the buffer size to 63K.

Creating a Pass 1 Listing (/D)

The /D option tells MASM to add a Pass 1 listing to the assembly listing file, making the assembly listing show the results of both assembler passes. A Pass 1 listing is typically used to locate phase errors. Phase errors occur when the assembler makes assumptions about the program in Pass 1 that are not valid in Pass 2. The /D option does not create a Pass 1 listing unless you also direct MASM to create an assembly listing. It directs the assembler to display error messages for both Pass 1 and Pass 2 of the assembly, even if no assembly listing is created. See Reading a Pass 1 Listing for more information about Pass 1 listings.

The following command line directs the assembler to create a Pass 1 listing for the source file file.asm. The file file.lst will contain both the first and second pass listings.

```
MASM  
    [args] /D file,,;
```

Defining Assembler Symbols

```
/Dsymbol [=value ]
```

The /D option, when given with a symbol argument directs MASM to define a symbol that can be used during the assembly as if it were defined as a text equate in the source file. Multiple symbols can be defined in a single command line.

The value can be any text string that does not include a space, comma, or semicolon. If no value is given, the symbol is assigned a null string.

As noted in The MASM Environment Variable, the version of the option using the equal sign cannot be stored in the MASM environment variable.

The following options define the symbol wide and gives it a null value. The symbol could then be used in the following conditional assembly block:

```
/Dwide /Dmode=3 file,,;  
  
    IFDEF wide  
        PAGE 50,132  
    ENDIF
```

When the symbol is defined in the command line, the listing file is formatted for a 132 column printer. When the symbol is not defined in the command line, the listing file is given the default width of 80 (see the description of the PAGE directive in Section 14).

The example also defines the symbol mode and gives it the value 3. The symbol could then be used in a variety of contexts, as shown below:

```
scrmode    IF      mode LT 15      ; Use in expression  
           DB      mode           ; Initialize to mode  
           ELSE  
scrmode    DB      15             ; Initialize to 15  
           ENDIF
```

Creating Code for a Floating Point Emulator (/E)

The /E option directs the assembler to generate data and code in the format expected by coprocessor emulator libraries. An emulator library uses the instructions of the 8087, 80287, or 80387 coprocessors, if a coprocessor is present; otherwise, the library emulates the coprocessor's activity.

Emulator libraries are only available with high level language compilers, including the Microsoft C, BASIC, FORTRAN, and Pascal compilers. The option cannot be used in stand alone assembler programs unless you write your own emulator library. You cannot simply link with the emulator library from a high level language, since these libraries require that the compiler start up code be executed.

The Microsoft high level language compilers allow you to use options to specify whether you want to use emulator code. If you link a high level language module prepared with emulator options with an assembler module that uses coprocessor instructions, you should use the /E option when assembling.

To the applications programmer, writing code for the emulator is like writing code for a coprocessor. The instruction sets are the same (except as noted in Section 21, Calculating with a Math Coprocessor). However, at run time the coprocessor instructions are used only if there is a coprocessor available on the machine. If there is no coprocessor, the slower code from the emulator library is used instead.

MASM

```
[args] /E /MX math.asm;
```

CL

```
[args] /FPi calc.c math
```

In the first command line, the source file math.asm is assembled with MASM by using the /E option. Then the CL program of the C compiler is used to compile the C source file calc.c with the /FPi option and finally to link the resulting object file (calc.obj) with math.obj. The compiler generates emulator code for floating point instructions. There are similar options for the FORTRAN, BASIC, and Pascal compilers.

Getting Command Line Help (/H)

The /H displays the command line syntax and all the MASM options on the screen. You should not give any filenames or other options with the /H option.

Setting a Search Path for Include Files (/I)

The /I option is used to set search paths for include files. You can set as many as 10 search paths by using the option for each path. The order of searching is the order in which the paths are listed in the command line. For example:

```
/I[sys] <inc> /I <macro> file;
```

These options might be used if the source file contains the following statement:

```
INCLUDE dos.inc
```

In this case, MASM searches for the file dos.inc first in directory <inc> on volume [sys], and then in directory <macro> on the current volume. If the file was not found in either of these directories, MASM looks next in the current directory and, finally, in any directories specified with the INCLUDE environment variable.

You should not specify a path name with the INCLUDE directive if you plan to specify search paths from the command line. For example, MASM only searches the specified path and ignores any search paths specified in the command line if the source file contains any of the following statements:

```
INCLUDE [sys] <macro> edos.inc  
INCLUDE <macro> edos.inc
```


Specifying Listing and Cross Reference Files

The /L option directs MASM to create a listing file even if one was not specified in the command line or in response to prompts. The /C option has the same effect for cross reference files. Files specified with these options always have the base name of the source file plus the extension .LST for listing files or .CRF for cross reference files. You cannot specify any other file name. Both options are provided for compatibility with the XENIX operating system.

The following options create file.lst and file.crf.

/L /C file;

It is equivalent entering MASM with the following:

file,,;

Specifying Case Sensitivity

The /ML option directs the assembler to make all names case sensitive. The /MX option directs the assembler to make public and external names case sensitive. The /MU option directs the assembler to convert all names to uppercase.

By default, MASM converts all names to uppercase. The /MU option (the default) is provided for compatibility with XENIX (which uses -MI by default) and to override options given in the environment variable.

If case sensitivity is turned on, all names that have the same spelling, but use letters of different cases, are considered different. For example, with the /ML option, DATA and data are different. They would also be different with the /MX option if they were declared external or public. Public and external names include any label, variable, or symbol names defined by using the EXTRN, PUBLIC, or COMM directives (see Section 10).

If you use the /ZI or /ZD option, the /MX, /ML, and /MU options affect the case of the symbolic data that will be available to a symbolic debugger.

The /ML and /MX options are typically used when object modules created with MASM are to be linked with object modules created by a case sensitive compiler such as the Microsoft C compiler. If case sensitivity is important, you should also use the linker /NOI option.

The following option is used to assemble a file with case sensitive public symbols.

/MX module;

Suppressing Tables in the Listing File

The /N option tells the assembler to omit all tables from the end of the listing file. If this option is not chosen, MASM includes tables of macros, structures, records, segments and groups, and symbols. The code portion of the listing file is not changed by the /N option.

/N file,;

Checking for Impure Code

The /P option directs MASM to check for impure code in the 80286 or 80386 privileged mode.

Code that moves data into memory with a CS: override is acceptable in real mode. However, such code may cause problems in protected mode. When the /P option is in effect, the assembler checks for these situations and generates an error if it encounters them.

Real and privileged modes are explained in Section 15. Versions of DOS available at release time do not support privileged mode.

This option is provided for XENIX compatibility and to warn about programming practices that will be illegal under protected mode operating systems.

The following example shows a CS override. If assembled with the /P option, an error is generated.

```
                .CODE
                .
                .
                .
addr            jmp          past          ; Don't execute data
past:          DW           ?             ; Allocate code space for data
                .
                .                         ; Calculate value of addr here
                .
                mov          cs:addr,si    ; Load register address
```

Controlling Display of Assembly Statistics

The /V and /T options specify the level of information displayed to the screen at the end of assembly. (V is a mnemonic for verbose; T is a mnemonic for terse.)

If neither option is given, MASM outputs a line telling the amount of symbol space free and the number of warnings and errors.

If the /V option is given, MASM also reports the number of lines and symbols processed.

If the /T option is given, MASM does not output anything to the screen unless errors are encountered. This option may be useful in batch or make files if you do not want the output cluttered with unnecessary messages.

If errors are encountered, they will be displayed whether these options are given or not. Appendix B describes the messages displayed after assembly.

Setting the Warning Level

The /W option sets the assembler warning level. For example:

```
/W{0 | 1 | 2}
```

MASM gives warning messages for assembly statements that are ambiguous or questionable but not necessarily illegal. Some programmers purposely use practices that generate warnings. By setting the appropriate warning level, they can turn off warnings if they are aware of the problem and do not wish to take action to remedy it.

MASM has three levels of errors, as shown in Table 4–2.

Table 4–2. Warning Levels

Level	Type	Description
0	Severe errors	Illegal statement
1	Serious warnings	Ambiguous statements or questionable programming practices
2	Advisory warnings	Statements that may produce inefficient code

The default warning level is 1. A higher warning level includes a lower level. Level 2 includes severe errors, serious warnings, and advisory warnings. If severe errors are encountered, no object file is produced.

The advisory warnings are listed below:

Number	Message
104	Jump within short distance
114	Operand size does not match segment word size
115	Address size does not match segment word size

The serious warnings are listed below:

Number	Message
1	Extra characters on line
16	Reserved word used as symbol: <i>name</i>
31	Operand types must match
57	Illegal size for operand
85	End of file, no END directive
101	Missing data; zero assumed
102	Segment near (or at) 64k limit

All other errors are severe.

Listing False Conditionals

The `/X` option directs MASM to copy to the assembly listing all statements forming the body of conditional assembly blocks whose condition is false. If you do not give the `/X` option in the command line, MASM suppresses all such statements. The `/X` option lets you display conditionals that do not generate code. Conditional assembly directives are explained in Section 14.

The `.LFCOND`, `.SFCOND`, and `.TFCOND` directives can override the effect of the `/X` option, as described in Section 14. The `/X` option does not affect the assembly listing unless you direct the assembler to create an assembly listing file.

The option

`/X file,;`

turns on listing of false conditionals when `file.asm` is assembled. Directives in the source file can override the `/X` option to change the status of false conditional listing.

Displaying Error Lines on the Screen

The `/Z` option directs MASM to display lines containing errors on the screen. The syntax is:

`/Z file;`

Normally when the assembler encounters an error, it displays only an error message describing the problem. When you use the `/Z` option in the command line, the assembler displays the source line that produced the error in addition to the error message. MASM assembles faster without the `/Z` option, but you may find the convenience of seeing the incorrect source lines worth the slight cost in processing speed.

Writing Symbolic Information to the Object File

The `/ZI` option directs MASM to write symbolic information to the object file. This option has no effect in CTOS.

There are two types of symbolic information available: line number data and symbolic data.

Line number data relates each instruction to the source line that created it.

Symbolic data specifies a size for each variable or label used in the program. This includes both public and nonpublic labels and variable names. Public symbols are discussed in Section 10.

The `/ZD` option writes line number information only to the object file. It can be used if you plan to debug with SYMDEB or if you want to see line numbers in map files. The `/ZI` option can also be used for these purposes, but it produces larger object files. If you do not have enough memory to debug a program with the

The option names `/ZI` and `/ZD` are similar to corresponding option names for recent versions of Microsoft compilers.

Reading Assembly Listings

This sections covers how to read assembly listings and includes the following subsections:

- Reading Code in a Listing
- Reading a Macro Table
- Reading a Structure and Record Table
- Reading a Segment and Group Table
- Reading a Symbol Table
- Reading Assembly Statistics
- Reading a Pass 1 Listing

MASM creates an assembly listing of your source file whenever you give an assembly listing file name on the MASM command line or in response to the MASM prompts. The assembly listing contains both the statements in the source file and the object code (if any) generated for each statement. The listing also shows the names and values of all labels, variables, and symbols in your source file.

The assembler creates tables for macros, structures, records, segments, groups, and other symbols. These tables are placed at the end of the assembly listing (unless you suppress them with the `/N` option). MASM lists only the types of symbols encountered in the program. For example, if your program has no macros, there will be no macro section in the symbol table. All symbol names will be shown in uppercase letters unless you use the `/ML` or `/MX` option to specify case sensitivity.

Reading Code in a Listing

The assembler lists the code generated from the statements of a source file. Each line has the following syntax:

[[linenumber]] offset [[code]] statement

The *linenumber* is the number of the line starting from the first statement in the assembly listing. Line numbers are produced only if you request a cross reference file. Line numbers in the listing do not always correspond to the same lines in the source file.

The *offset* is the offset from the beginning of the current segment to the code. If the statement generates code or data, code shows the numeric value in hexadecimal if the value is known at assembly time. If the value is calculated at run time, MASM indicates what action is necessary to compute the value. The statement is the source statement shown exactly as it appears in the source file, or as expanded by a macro.

If any errors occur during assembly, each error message and error number will appear directly below the statement where the error occurred. Refer to Appendix B for a list of MASM errors and a discussion of the format in which errors are displayed. The following example shows an error line and message:

```
       71 0012 E8 001C R                               call      doit
test.ASM(46): error A2071: Forward needs override or FAR
```

Note that number 46 in the error message is the source line where the error occurred. Number 71 on the code line is the listing line where the error occurred. These lines will seldom be the same.

The assembler uses the symbols and abbreviations in Table 4–3 to indicate addresses that need to be resolved by the linker or values that were generated in a special way.

Table 4–3. Symbols and Abbreviations in Listings

Character	Meaning
R	Relocatable address (linker must resolve)
E	External address (linker must resolve)
---	Segment/group address (linker must resolve)
=	EQU or equal sign (=) directive
<i>nn</i> :	Segment override in statement
<i>nn</i> /	REP or LOCK prefix instruction
<i>nn</i> [<i>xx</i>]	DUP expression: <i>nn</i> copies of the value <i>xx</i>
<i>n</i>	Macro expansion nesting level (+ if more than nine)
C	Line from INCLUDE file
I	80386 size or address prefix

The sample listing shown in Figure 4–1 is produced using the /ZI option. A cross reference file is specified so that line numbers will appear in the listing. The command line is:

```
MASM
    [args] /ZI listdemo,,;
```

Figure 4–1 shows the code portion of the resulting listing. The tables normally seen at the end of the listing are explained later.

CTOS Microsoft (R) Macro Assembler Version 5.1.1

5/23/91 10:46:45

Listing features demo

Page 1-1

```

1          ; Declare the OS and object module procedures as external,
2          ; accessible by FAR CALLS
3
4          EXTRN ErrorExit: FAR
5
6          PAGE 67,132
7          TITLE Listing features demo
8          INCLUDE string.mac
9          C      StrAlloc      MACRO name,text
10         C      name      DB      &text
11         C      DB      13d,10d
12         C      l&name      EQU      $-name
13         C      ENDM
14
15 = 0080          larg      EQU      80h
16
17 .MODEL small
18
19 .STACK 256
20
21         color      RECORD      b:1,r:3=1,i:1=1,f:3=7
22
23         date      STRUC
24 0000          05      month      DB      5
25 0001          07      day      DB      7
26 0002          07C7    year      DW      1991
27 0004          date      ENDS
28
29         .DATA
30 0000          1F      text      color      <>
31 0001          05      today      date      <5,22,1991>
32 0002          16
33 0003          C707
34
35 0005          0064[      buffer      DW      100 DUP(?)
36          ]
37
38
39
40          StrAlloc ending, <"Finished.">
41 00CD 46 69 6E 69 73 68          1 ending      DB      "Finished."
42 00D6 0D 0A 1 DB 13d,10d
43

```

Figure 4-1. CTOS Microsoft Macro Assembler Listing

```

44                                     .CODE
45 0000 B8 ---- R      Start:  mov     ax, @DATA
46 0003 8E D8          mov     ds,ax
47
48 0005 B8 0063 mov     ax,'c'
49 0008 26: 8B 0E 0080    mov     cx,es:larg
50 000D BF 0052          mov     di,82
51 0010 F2/ AE          repne   scasd
52 0012 57 push di
53
54                                     EXTRN  work:NEAR
55 0013 E8 0000 E      call    work
56
57 0016 B8 170C          mov     ax,4C001istdemo.ASM(44): error
58 0019 9A 0000 ---- E      call    A2107: Illegal digit in number
59                               ErrorExit
60                               END      Start

```

Figure 4–1. CTOS Microsoft Macro Assembler Listing (continued)

Reading a Macro Table

A macro table at the end of a listing file gives, in alphabetical order, the names and sizes (in lines) of all macros called or defined in the source file. For example:

Macros:

N a m e	Lines
STRALLOC	3

Reading a Structure and Record Table

All structures and records declared in the source file are given at the end of the listing file. The names are listed in alphabetical order. Each name is followed by the fields in the order in which they are declared. For example:

Structures and Records:

Name	Width Shift	# fields	
		Width	Mask
COLOR	0008	0004	
B	0007	0001	0080 0000
R	0004	0003	0070 0010
I	0003	0001	0008 0008
F	0000	0003	0007 0007
DATE	0004	0003	
MONTH	0000		
DAY	0001		
YEAR	0002		

The first row of headings only applies to the record or structure itself. For a record, the “Width” column shows the width in bits while the column “# fields” tells the total number of fields.

The second row of headings applies only to fields of the record or structure. For records, the “Shift” column lists the offset (in bits) from the low order bit of the record to the low order bit in the field. The “Width” column lists the number of bits in the field. The “Mask” column lists the maximum value of the field, expressed in hexadecimal. The “Initial” column lists the initial value of the field, if any. For each field, the table shows the mask and initial values as if they were placed in the record and all other fields were set to 0.

For a structure, the “Width” column lists the size of the structure in bytes. The “# fields” column lists the number of fields in the structure. Both values are in hexadecimal.

For structure fields, the “Shift” column lists the offset in bytes from the beginning of the structure to the field. This value is in hexadecimal. The other columns are not used.

Reading a Segment and Group Table

Segments and groups used in the source file are listed at the end of the program with their size, align type, combine type, and class. If you used simplified segment directives in the source file, the actual segment names generated by MASM will be listed in the table.

Segments and Groups:

Name	Length	Align	Combine	Class
DGROUP	GROUP			
_DATA	00D8	WORD	PUBLIC	'DATA'
STACK	0100	PARA	STACK	'STACK'
_TEXT	001E	WORD	PUBLIC	'CODE'

The “Name” column lists the names of all segments and groups. Segment and group names are given in alphabetical order, except that the names of segments belonging to a group are placed under the group name in the order in which they were added to the group.

The “Size” column lists the byte size (in hexadecimal) of each segment. The size of groups is not shown.

The “Align” column lists the align type of the segment.

The “Combine” column lists the combine type of the segment. If no explicit combine type is defined for the segment, the listing shows NONE, representing the private combine type. If the “Align” column contains AT, the “Combine” column contains the hexadecimal address of the beginning of the segment.

The “Class” column lists the class name of the segment. For a complete explanation of the align, combine, and class types, see Section 7.

Reading a Symbol Table

All symbols (except names for macros, structures, records, and segments) are listed in a symbol table at the end of the listing. For example:

Symbols:

Name	Type	Value	Attr
B	0007		
BUFFER	L WORD	0005	_DATA
Length = 0064			
ENDING	L BYTE	00CD	_DATA
ERREXIT	L FAR	0000	External
F		0000	
I		0003	
LARG	NUMBER	0080	
LENDING	NUMBER	000B	
R		0004	
START	L NEAR	0000	_TEXT
TEXT	L BYTE	0000	_DATA
TODAY	L DWORD	0001	_DATA
WORK	L NEAR	0000	_TEXT
External			
@CODE	TEXT	__TEXT	
@CODESIZE	TEXT	0	
@CPU	TEXT	0101h	
@DATASIZE	TEXT	0	
@FILENAME	TEXT	listdemo	
@VERSION	TEXT	511	

The “Name” column lists the names in alphabetical order. The “Type” column lists each symbol’s type. A type is given as one of the following:

Type	Definition
L NEAR	A near label
L FAR	A far label
N PROC	A near procedure label
F PROC	A far procedure label
NUMBER	An absolute label
ALIAS	An alias for another symbol
OPCODE	An equate for an instruction opcode
TEXT	A text equate
BYTE	One byte
WORD	One word (two bytes)
DWORD	Doubleword (four bytes)
FWORD	Farword (six bytes)
QWORD	Quadword (eight bytes)
TBYTE	Ten bytes
<i>number</i>	Length in bytes of a structure variable

The length of a multiple element variable such as an array or string is the length of a single element, not the length of the entire variable. For example, string variables are always shown as L BYTE.

If the symbol represents an absolute value defined with an EQU or equal sign (=) directive, the Value column shows the symbol’s value. The value may be another symbol, a string, or a constant numeric value (in hexadecimal), depending on whether the type is ALIAS, TEXT, or NUMBER. If the type is OPCODE, the Value column will be blank. If the symbol represents a variable, label, or procedure, the Value column shows the symbol’s hexadecimal offset from the beginning of the segment in which it is defined.

The “Attr” column shows the attributes of the symbol. The attributes include the name of the segment (if any) in which the symbol is defined, the scope of the symbol, and the code length. A symbol’s scope is given only if the symbol is defined using the EXTRN and PUBLIC directives. The scope can be EXTERNAL, GLOBAL, or COMMUNAL. The code length (in hexadecimal) is given only for procedures. The “Attr” column is blank if the symbol has no attribute.

The text equates shown at the end of the sample table are the ones defined automatically when you use simplified segment directives (see Section 7).

Reading Assembly Statistics

Data on the assembly, including the number of lines and symbols processed and the errors or warnings encountered, are shown at the end of the listing. See Appendix B for further information on this data. For example:

```
52 Source    Lines
55 Total     Lines
36 Symbols
33140 + 155566 Bytes symbol space free
0 Warning    Errors
1 Severe     Errors
```

Reading a Pass 1 Listing

When you specify the /D option in the MASM command line, the assembler puts a Pass 1 listing in the assembly listing file. The listing file shows the results of both assembler passes. Pass 1 listings are useful in analyzing phase errors.

The following example illustrates a Pass 1 listing for a source file that assembled without error on the second pass.

```
0017 7E 00          jle    label1
PASS_CMP.ASM(20) : error 9 : Symbol not defined LABEL1
0019 BB 1000        mov    bx,4096
001C                label1:
```

During Pass 1, the JLE instruction to a forward reference produces an error message, and the value 0 is encoded as the operand. MASM displays this error because it has not yet encountered the symbol label1.

Later in Pass 1, label1 is defined. Therefore, the assembler knows about label1 on Pass 2 and can fix the Pass 1 error. The Pass 2 listing is shown below:

```
0017 7E 03          jle    label1
0019 BB 1000        mov    bx,4096
001C                label1:
```

The operand for the JLE instruction is now coded as 3 instead of 0 to indicate that the distance of the jump to label1 is three bytes.

Since MASM generated the same number of bytes for both passes, there was no error. Phase errors occur if the assembler makes an assumption on Pass 1 that it cannot change on Pass 2. If you get a phase error, you can examine the Pass 1 listing to see what assumptions the assembler made.

Section 5

Using CREF

The CTOS Microsoft Cross Reference Utility (CREF) creates a cross reference listing of all symbols in an assembly language program. A cross reference listing is an alphabetical list of symbols in which each symbol is followed by a series of line numbers. The line numbers indicate the lines in the source program that contain a reference to the symbol.

CREF is intended for use as a debugging aid to speed up the search for symbols encountered during a debugging session. The cross reference listing, together with the symbol table created by the assembler, can make debugging and correcting a program easier. The following subsections describe how to use CREF:

- Using CREF
- Reading cross reference listings

Using CREF

This subsection describes how to use CREF and contains the following:

- Using a Command Line to Create a Cross Reference Listing
- Using Prompts to Create a Cross Reference Listing

CREF creates a cross reference listing for a program by converting a binary cross reference file, produced by the assembler, into a readable ASCII file. You create the cross reference file by supplying a cross reference file name when you invoke the assembler. See Section 4 for more information on creating a binary cross reference file. To create. You create the cross reference listing by invoking CREF and supplying the name of the cross reference file.

Using a Command Line to Create a Cross Reference Listing

To convert a binary cross reference file created by MASM into an ASCII cross reference listing, type CREF. Then enter the names of the files you want to process on the parameter line.

CREF

[args] *crossreferencefile* [[*crossreferencelisting*] [;]

The *crossreferencefile* is the name of the cross reference file created by MASM, and the *crossreferencelisting* is the name of the readable ASCII file you want to create.

If you do not supply file name extensions when you name the files, CREF automatically provides .CRF for the cross reference file and .REF for the cross reference listing file. If you do not want these extensions, you must supply your own.

To select a default file name for the listing file type a semicolon (;) immediately after *crossreferencefile*.

You can specify a directory or disk drive for either of the files. You can also name output devices such as CON (display console) and PRN (printer).

When CREF finishes creating the cross reference listing file, it displays the number of symbols processed.

Enter the following command line:

The following example converts the cross reference file test.crf to the cross reference listing file test.ref.

CREF

[args] test.crf,test.ref

It is equivalent to entering the following:

CREF

[args] test,test

or

CREF

[args] test;

The following example directs the cross reference listing to the screen. If CREF is entered with the following parameters, no file is created.

```
CREF  
[args] test,con
```

Using Prompts to Create a Cross Reference Listing

To direct CREF to prompt you for the files it needs, start CREF with just the command name. CREF prompts you for the input it needs by displaying the following lines, one at a time:

```
Cross Reference [.CRF]:  
Listing [filename.REF]:
```

The prompts correspond to the fields of CREF command lines. CREF waits for you to respond to each prompt before printing the next one. You must type a cross reference file name (though the extension is optional) at the first prompt. For the second prompt, you can either type a file name or press the RETURN key to accept the default displayed in brackets after the prompt.

After you have answered the last prompt and pressed the RETURN key, CREF reads the cross reference file and creates the new listing. It also displays the number of symbols in the cross reference file.

Reading Cross Reference Listings

The cross reference listing contains the name of each symbol defined in your program. Each name is followed by a list of line numbers representing the lines in the listing file in which a symbol is defined or used. Line numbers in which a symbol is defined are marked with a number sign (#).

Each page in the listing begins with the title of the program. The title is the name or string defined by the TITLE directive in the source file (see Section 14).

The next three code samples illustrate source, listings, and cross reference files for a program. The source file `hello.asm` is shown below:

```
PAGE 64,132

        TITLE HELLO
        .MODEL small
        .STACK 100h

        .DATA

PUBLIC  message, lmessage
message DB      "Hello, world.",13,10
lmessage EQU    $ - message

        .CODE

Start:
    EXTRN    display: PROC
            call    display

    EXTRN    ErrorExit: FAR
            push    ax
            call    ErrorExit

        END      Start
```

To assemble the program and create a cross reference file, run MASM with the following parameter line:

`hello,;,;`

The listing file `hello.lst` produced by this assembly is shown in Figure 5–1:

CTOS Microsoft (R) Macro Assembler Version 5.1.1		
5/23/91 14:17:45		
HELLO		Page 1-1
1	PAGE 64,132	
2		TITLE HELLO
3		.MODEL small
4		
5		.STACK 100h
6		
7		.DATA
8		

Figure 5–1. Example Assembly Listing

```

9
10 0000 48 65 6C 6C 6F 2C    message DB    PUBLIC message, lmessage
11 20 77 6F 72 6C 64        "Hello, world." ,13,10
12 2E 0D 0A
13 = 000F                    lmessage EQU    $ -message
14
15                            .CODE
16
17 0000                        Start:
18                            EXTRN    display: PROC
19 0000 E8 0000 E             call    display      20
21                            EXTRN    ErrorExit: FAR
22 0003 50                   push    ax
23 0004 9A 0000 ---- E       call    ErrorExit
24
25                            END      Start

```

CTOS Microsoft (R) Macro Assembler Version 5.1.1

5/23/91 14:17:45

HELLO

Symbols-1

Segments and Groups:

	N a m e	Length	Align	Combine
Class				
DGROUP	GROUP		
_DATA	000F	WORD	PUBLIC
'DATA'				
STACK	0100	PARA	STACK
'STACK'				
_TEXT	0009	WORD	PUBLIC
'CODE'				

..

Figure 5-1. Example Assembly Listing (continued)

Symbols:

Name	Type	Value	Attr
DISPLAY External	L NEAR	0000	_TEXT
ERROREXIT External	L FAR	0000	
LMESSAGE Global	NUMBER	000F	
MESSAGE Global	L BYTE	0000	_DATA
START	L NEAR	0000	_TEXT
@CODE	TEXT	_TEXT	
@CODESIZE	TEXT	0	
@CPU	TEXT	0101h	
@DATASIZE	TEXT	0	
@FILENAME	TEXT	hello	
@VERSION	TEXT	511	
23 Source Lines			
23 Total Lines			
22 Symbols			
33350 + 163789 Bytes symbol space free			
0 Warning Errors			
0 Severe Errors			

Figure 5–1. Example Assembly Listing (continued)

To create a cross reference listing of the file hello.crf, enter the following command:

CREF

[args] hello;

The resulting cross reference listing file hello.ref is shown in Figure 5-2.

CTOS Microsoft Cross-Reference Version 5.1.1

Thu May 23

14:19:57 1991

HELLO

Symbol Cross-Reference	(# definition, + modification)	
------------------------	--------------------------------	--

Cref-1

@CPU	1#	
@VERSION	1#	
CODE	15	
DATA	7	
DISPLAY.	18#	19
ERROREXIT.	21#	23
LMESSAGE	9	13#
MESSAGE.	9	10#
STACK.	5#	5
START.	17#	25
_DATA.	7#	
_TEXT.	15#	

12 Symbols

Figure 5-2. Example Cross Reference Listing

Notice that line numbers in the listing and cross reference listing files may not identify corresponding lines in the source file.

Section 6

Writing Source Code

Assembly language programs are written as source files, which can then be assembled into object files by MASM. Object files can then be processed and combined with LINK to form executable files.

Source files are made up of assembly language statements. Statements are in turn made up of mnemonics, operands, and comments. This section describes how to write assembly language statements. Symbol names and constants are explained. It also tells you how to start and end assembly language source files. The information is contained in the following subsections:

- Writing Assembly Language Statements
- Assigning Names to Symbols
- Constants
- Defining Default Assembly Behavior
- Ending a Source File

Writing Assembly Language Statements

A statement is a combination of mnemonics, operands, and comments that defines the object code to be created at assembly time. Each line of source code consists of a single statement. Multiline statements are not allowed. Statements must not have more than 128 characters. Statements can have up to four fields, as shown below:

`[[name]] [[operation]] [[operands]] [;comment]`

The fields are described below, starting with the leftmost field:

Field	Purpose
name	Labels the statement so that the statement can be accessed by name in other statements
operation	Defines the action of the statement
operands	Defines the data to be operated on by the statement
comment	Describes the statement without having any effect on assembly

All fields are optional, although the operand or name fields may be required if certain directives or instructions are given in the operation field. A blank line is simply a statement in which all fields are blank. A comment line is a statement in which all fields except the comment are blank.

Statements can be entered in uppercase or lowercase letters. Sample code in this manual uses uppercase letters for directives, hexadecimal letter digits, and segment definitions. Your code will be clearer if you choose a case convention and use it consistently. Each field (except the comment field) must be separated from other fields by a space or tab character. That is the only limitation on structure imposed by MASM. For example, the following code is legal:

```
; Declare the OS and object module procedures as external,
; accessible by FAR CALLs
EXTRN WriteBsRecord: FAR, ErrorExit: FAR
TITLE HELLO
.MODEL small;Use small model
.DATA
message DB "Hello, world.",13,10;Message to be written
lmessage EQU $-message;Length of message
;We write to video using SAM's pre-opened bytestream
;which is located in the data segment. It is important
;to locate this declaration within the DATA SEGMENT as
;shown here.
EXTRN bsVid: BYTE
cbWrittenRet DW ?
.286 ;allow protected mode opcodes
.CODE
Start:
;Here we will print the message using CTOS call WriteBsRecord
push ds;1st arg is pbsVid
push OFFSET bsVid
push ds;2nd arg is prgcsMsg
push OFFSET message
push lmessage ;3rd arg is cbMsg
```

```
push ds;4th arg is pcbWrittenRet
push OFFSET cbWrittenRet
call WriteBsRecord;make the call
push ax;AX contains "rcode"
call ErrorExit
END Start
```

However, the code is much easier to interpret if each field is assigned a specified tab position and a standard convention is used for capitalization. The example program in Section 3 is the same as the example above except for the conventions used.

Using Mnemonics and Operands

Mnemonics are the names assigned to commands that tell either the assembler or the processor what to do. There are two types of mnemonics: directives and instructions.

Directives give directions to the assembler. They specify the manner in which the assembler is to generate object code at assembly time.

Instructions give directions to the processor. At assembly time, they are translated into object code. At run time, the object code controls the behavior of the processor.

Operands define the data that is used by directives and instructions. They can be made up of symbols, constants, expressions, and registers. Assigning Names to Symbols and Constants cover symbol names and constants. Operands, expressions, and registers are discussed throughout the manual, but particularly in Section 11 and Section 16.

Writing Comments

Comments are descriptions of the code. They are for documentation only and are ignored by the assembler.

Any text following a semicolon is considered a comment. Comments commonly start in the column assigned for the comment field, or in the first column of the source code. The comment must follow all other fields in the statement.

You can specify multiline comments with multiple comment statements or with the `COMMENT` directive.

```
COMMENT delimiter [text ]  
text  
delimiter [text ]
```

All *text* between the first *delimiter* and the line containing a second *delimiter* is ignored by the assembler. The *delimiter* character is the first nonblank character after the `COMMENT` directive. The *text* includes the comments up to and including the line containing the next occurrence of the delimiter.

```
COMMENT + The plus  
          sign is the delimiter. The  
          assembler ignores the statement  
          following the last delimiter +  
mov      ax,1 (ignored)
```

Assigning Names to Symbols

A symbol is a name that represents a value. Symbols are one of the most important elements of assembly language programs. Elements that must be represented symbolically in assembly language source code include variables, address labels, macros, segments, procedures, records, and structures. You can also represent constants, expressions, and strings symbolically.

Symbol names are combinations of letters (both uppercase and lowercase), digits, and special characters. MASM recognizes the following character set:

Letters and digits:
A–Z a–z 0–9

Special Characters:

? @ _ \$: . [] () < > { } + - / * & ! ' ~ | \ = # ^ ; , ' .

Letters, digits, and some characters can be used in symbol names, but some restrictions on how certain characters can be used or combined are listed below:

- A name can have any combination of uppercase and lowercase letters. All lowercase letters are converted to uppercase by the assembler, unless the /ML assembly option is used, or unless the name is declared with a PUBLIC or EXTRN directive and the /MX option is used.
- Digits may be used within a name, but not as the first character.
- A name can be given any number of characters, but only the first 31 are used. All other characters are ignored.
- The following characters may be used at the beginning of a name or within a name: underscore (_), question mark (?), dollar sign (\$), and at sign (@).
- The period (.) is an operator and cannot be used within a name, but it can be used as the first character of a name.
- A name may not be the same as any reserved name. Note that two special characters, the question mark (?) and the dollar sign (\$), are reserved names and therefore cannot stand alone as symbol names.

A reserved name is any name with a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and operator names. All uppercase and lowercase letter combinations of these names are treated as the same name.

Table 6–1 lists names that are always reserved by the assembler. Using any of these names for a symbol results in an error.

Table 6-1. Reserved Names

\$.DATA	.ERRNDEF	.LALL	REPT
*	.DATA?	.ERRNZ	LE	.SALL
+	DB	EVEN	LENGTH	SEG
-	DD	EXITM	.LFCOND	SEGMENT
.	DF	EXTRN	.LIST	.SEQ
/	DOSSEG	FAR	LOCAL	.SFCOND
=	DQ	.FARDATA	LOW	SHL
?	DS	.FARDATA?	LT	SHORT
[]	DT	FWORD	MACRO	SHR
.186	DW	GE	MASK	SIZE
.286	DWORD	GROUP	MOD	.STACK
.286P	ELSE	GT	.MODEL	STRUC
.287	END	HIGH	NAME	SUBTTL
.386	ENDIF	IF	NE	TBYTE
.386P	ENDM	IF1	NEAR	.TFCOND
.387	ENDP	IF2	NOT	THIS
.8086	ENDS	IFB	OFFSET	TITLE
.8087	EQ	IFDEF	OR	TYPE
ALIGN	EQU	IFDIF	ORG	.TYPE
ALPHA	.ERR	IFE	OUT	WIDTH
AND	.ERR1	IFIDN	PAGE	WORD
ASSUME	.ERR2	IFNB	PROC	.XALL
BYTE	.ERRB	IFNDEF	PTR	.XCREF
.CODE	.ERRDEF	INCLUDE	PUBLIC	.XLIST
COMM	.ERRDIF	INCLUDELIB	PURGE	XOR
COMMENT	.ERRE	IRP	QWORD	
.CONST	.ERRIDN	IRPC	.RADIX	
.CREF	.ERRNB	LABEL	RECORD	

In addition to these names in Table 6–1, instruction mnemonics and register names are considered reserved names. These vary depending on the processor directives given in the source file. For example, the register name EAX is a reserved word with the .386 directive but not with the .286 directive. Defining Default Assemble Behavior describes processor directives. Register names are listed in Section 16.

Constants

You can use constants in source files to specify numbers or strings that are set or initialized at assembly time. MASM recognizes four types of constant values:

- Integers
- Packed binary coded decimals
- Real numbers
- Strings

Integer Constants

Integer constants represent integer values. They can be used in a variety of contexts in assembly language source code. For example, they can be used in data declarations and equates, or as immediate operands.

Packed decimal integers are a special kind of integer constant that can only be used to initialize binary coded decimal (BCD) variables. They are described in Packed Binary Coded Decimal Constants and in Section 8.

Integer constants can be specified in binary, octal, decimal, or hexadecimal values. Table 6–2 shows the legal digits for each of these radices. For the hexadecimal radix, the digits can be either uppercase or lowercase letters.

Table 6–2. Digits Used with Each Radix

Name	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

The radix for an integer can be defined for a specific integer by using radix specifiers, or a default radix can be defined globally with the `.RADIX` directive.

Specifying Integers with Radix Specifiers

The radix for an integer constant can be given by putting one of the following radix specifiers after the last digit of the number:

Radix	Specifier
Binary	B
Octal	Q or O
Decimal	D
Hexadecimal	H

Radix specifiers can be given in either uppercase or lowercase letters; sample code in this manual uses lowercase letters.

Hexadecimal numbers must always start with a decimal digit (0–9). If necessary, put a leading 0 at the left of the number to distinguish between symbols and hexadecimal numbers that start with a letter. For example, 0ABCh is interpreted as a hexadecimal number, but ABCh is interpreted as a symbol. The hexadecimal digits A through F can be either uppercase or lowercase letters. Sample code in this manual uses uppercase letters.

If no radix is given, the assembler interprets the integer by using the current default radix. The initial default radix is decimal, but you can change the default with the `.RADIX` directive. Following are definitions using different radices.

```
n360    EQU    01011010b + 132q + 5Ah  + 90d ; 4 * 90
n60     EQU    00001111b + 17o  +  0Fh + 15d ; 4 * 15
```

Setting the Default Radix

The `.RADIX` directive sets the default radix for integer constants in the source file. The syntax is:

`.RADIX expression`

The *expression* must evaluate to a number in the range 2–16. It defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base.

Numbers given in *expression* are always considered decimal, regardless of the current default radix. The initial default radix is decimal. For example:

```
.RADIX    16      ; Set default radix to hexadecimal
.RADIX     2      ; Set default radix to binary
```

The `.RADIX` directive does not affect real numbers initialized as variables with the `DD`, `DQ`, or `DT` directive. Initial values for real number variables declared with these directives are always evaluated as decimal unless a radix specifier is appended.

Also, the `.RADIX` directive does not affect the optional radix specifiers, `B` and `D`, used with integer numbers. When the letters `B` or `D` appear at the end of any integer, they are always considered to be a radix specifier even if the current radix is 16.

For example, if the input radix is 16, the number `0ABCD` is interpreted as `0ABC` decimal, an illegal number, instead of as `0ABCD` hexadecimal, as intended. Type `0ABCDh` to specify `0ABCD` in hexadecimal. Similarly, the number `11B` is treated as 11 binary, a legal number, but not as `11B` hexadecimal as intended. Type `11Bh` to specify `11B` in hexadecimal.

Packed Binary Coded Decimal Constants

When an integer constant is used with the `DT` directive, the number is interpreted by default as a packed binary coded decimal number. You can use the `D` radix specifier to override the default and initialize 10 byte integers as binary format integers.

The syntax for specifying binary coded decimals is exactly the same as for other integers. However, MASM encodes binary coded decimals in a completely different way as shown in the following example:

```
positive    DT    1234567890    ; Encoded as 00000000001234567890h
negative    DT   -1234567890    ; Encoded as 80000000001234567890h
```

See Section 8 for complete information on storage of binary coded decimals.

Real Number Constants

A real number is a number consisting of an integer part, a fractional part, and an exponent. Real numbers are usually represented in decimal format.

`[[+ | -] integer.fraction [E [+ | -] exponent]`

The *integer* and *fraction* parts combine to form the value of the number. This value is stored internally as a unit and is called the mantissa. It may be signed. The optional *exponent* follows the exponent indicator (E). It represents the magnitude of the value, and is stored internally as a unit. If no *exponent* is given, 1 is assumed. If an exponent is given, it may be signed.

During assembly, MASM converts real number constants given in the decimal format to a binary format. The sign, exponent, and mantissa of the real number are encoded as bit fields within the number. See Section 8 for an explanation of how real numbers are encoded.

You can specify the encoded format directly using hexadecimal digits (0–9 or A–F). The number must begin with a decimal digit (0–9) and cannot be signed. It must be followed by the real number designator (R). This designator is used the same as a radix designator except it specifies that the given hexadecimal number should be interpreted as a real number.

Real numbers can only be used to initialize variables with the DD, DQ, and DT directives. They cannot be used in expressions. The maximum number of digits in the number and the maximum range of exponent values depend on the directive. The number of digits for encoded numbers used with DD, DQ, and DT must be 8, 16, and 20 digits, respectively. (If a leading 0 is supplied, the number must be 9, 17, or 21 digits.) See Section 8 for an explanation of how real numbers are encoded.

Note: *Real numbers are encoded differently depending upon whether you use the .MSFLOAT directive. By default, real numbers are encoded in the IEEE format. This is a change from previous versions, which assembled real numbers by default in the Microsoft Binary format. The .MSFLOAT directive overrides the default and specifies Microsoft Binary format. See Section 8 for a description of these formats.*

```

; Real numbers
shrt      DD      25.23
long      DQ      2.523E1
ten_byte  DT      2523.0E-2

; Assumes .MSFLOAT
mbshort   DD      81000000r      ;1.0 as Microsoft Binary short
mblong    DQ      8100000000000000r ;1.0 as Microsoft Binary long

; Assumes default IEEE format
ieeeshort DD      3F800000r      ; 1.0 as IEEE short
ieelong   DQ      3FF0000000000000r ; 1.0 as IEEE long

; The same regardless of processor directives
temporary DT      3FFF8000000000000000r ; 1.0 as 10 byte temporary real
```

String Constants

A string constant consists of one or more ASCII characters enclosed in single or double quotation marks.

'characters'
"characters"

String constants are case sensitive. A string constant consisting of a single character is sometimes called a character constant.

Single quotation marks must be encoded twice when used literally within string constants that are also enclosed by single quotation marks. Similarly, double quotation marks must be encoded twice when used in string constants that are also enclosed by double quotation marks.

```
char      DB      'a'
char2     DB      a
message   DB      This is a message.
warn      DB      'Can''t find file.'      ; Can't find file.
warn2     DB      Can't find file.          ; Can't find file.
string    DB      This value not found.    ; This value not found.
string2   DB      'This value not found.' ; This value not found.
```

Defining Default Assembly Behavior

Since the assembler processes sequentially, any directives that define the behavior of the assembler for sections of code or for the entire source file must come before the sections affected by the directive.

There are three types of directives that may define behavior for the assembly:

- The `.MODEL` directive defines the memory model.
- Processor directives define the processor and coprocessor.
- The `.MSFLOAT` directive and the coprocessor directives define how floating point variables are encoded.

These directives are optional. If you do not use them, MASM makes default assumptions. However, if you do use them, you must put them before any statements that will be affected by them.

The `.MSFLOAT` and `.MODEL` directives affect the entire assembly and can only occur once in the source file. Normally they should be placed at the beginning of the source file.

The `.MODEL` directive is part of the new system of simplified segment directives implemented in MASM. This directive is explained in Section 7.

The `.MSFLOAT` directive disables all coprocessor instructions and specifies that initialized real number variables be encoded in the Microsoft Binary format. Without this directive, initialized real number variables are encoded in the IEEE format. This is a change from previous versions of the assembler, which used Microsoft Binary format by default and required a coprocessor directive or the `/R` option to specify IEEE format. `.MSFLOAT` must be used for programs that require real number data in the Microsoft Binary format. Section 8 describes real number data formats and the factors to consider in choosing a format.

Processor and coprocessor directives define the instruction set that is recognized by MASM. They are listed and explained below:

Directive	Description
<code>.8086</code>	<p>The <code>.8086</code> directive enables assembly of instructions for the 8086 and 8088 processors and the 8087 coprocessor. It disables assembly of the instructions unique to the 80186, 80286, and 80386 processors.</p> <p>This is the default mode and is used if no instruction set directive is specified. Using the default instruction set ensures that your program can be used on all 8086 family processors. However, if you choose this directive, your program will not take advantage of the more powerful instructions available on more advanced processors.</p>
<code>.186</code>	<p>The <code>.186</code> directive enables assembly of the 8086 processor instructions, 8087 coprocessor instructions, and the additional instructions for the 80186 processor.</p>
<code>.286</code>	<p>The <code>.286</code> directive enables assembly of the 8086 instructions plus the additional nonprivileged instructions of the 80286 processor. It also enables 80287 coprocessor instructions. If privileged instructions were previously enabled, the <code>.286</code> directive disables them.</p> <p>This directive should be used for programs that will be executed only by an 80186, 80286, or 80386 processor. For compatibility with previous versions of MASM, the <code>.286C</code> directive is also available. It is equivalent to the <code>.286</code> directive.</p>

- .286P** This directive is equivalent to the **.286** directive except that it also enables the privileged instructions of the 80286 processor. This does not mean that the directive is required if the program will run in protected mode; it only means that the directive is required if the program uses the instructions that initiate and manage privileged mode processes. These instructions (see Section 22) are normally used only by systems programmers.
- .386** The **.386** directive enables assembly of the 8086 and the nonprivileged instructions of the 80286 and 80386 processors. It also enables 80387 coprocessor instructions. If privileged instructions were previously enabled, this directive disables them.
- This directive should be used for programs that will be executed only by an 80386 processor.
- .386P** This directive is equivalent to the **.386** directive except that it also enables the privileged instructions of the 80386 processor.
- .8087** The **.8087** directive enables assembly of instructions for the 8087 math coprocessor and disables assembly of instructions unique to the 80287 coprocessor. It also specifies the IEEE format for encoding floating point variables.
- This is the default mode and is used if no coprocessor directive is specified. This directive should be used for programs that must run with either the 8087, 80287, or 80387 coprocessors.
- .287** The **.287** directive enables assembly of instructions for the 8087 floating point coprocessor and the additional instructions for the 80287. It also specifies the IEEE format for encoding floating point variables.
- Coprocessor instructions are optimized if you use this directive rather than the **.8087** directive. Therefore, you should use it if you know your program will never need to run under an 8087 processor. See Section 21 for an explanation.

.387 The .387 directive enables assembly of instructions for the 8087 and 80287 floating point coprocessors and the additional instructions and addressing modes for the 80387. It also specifies the IEEE format for encoding floating point variables.

If you do not specify any processor directives, MASM uses the following defaults:

- 8086/8088 processor instruction set
- 8087 coprocessor instruction set
- IEEE format for floating point variables

Normally the processor and coprocessor directives can be used at the start of the source file to define the instruction sets for the entire assembly. However, it is possible to use different processor directives at different points in the source file to change assumptions for a section of code. For instance, you might have processor specific code in different parts of the same source file. You can also turn privileged instructions on and off or allow unusual combinations of the processor and coprocessor.

There are two limitations on changing the processor or coprocessor:

- The directives must be given outside segments. You must end the current segment, give the processor directive, and then open another segment. See Section 7 for an example of changing the processor directives with simplified segment directives.
- You can specify a lower level coprocessor with a higher level coprocessor, but an error message will be generated if you try to specify a lower level processor with a higher level coprocessor.

The coprocessor directives have the opposite effect of the `.MSFLOAT` directive. `.MSFLOAT` turns off coprocessor instruction sets and enables the Microsoft Binary format for floating point variables. Any coprocessor instruction turns on the specified coprocessor instruction set and enables IEEE format for floating point variables. Following is an example of using coprocessor instructions:

```
; .MSFLOAT affects the whole source file
      .MSFLOAT
      .8087                ; Ignored

;Legal – use 80386 and 80287
      .386
      .287

;Illegal – can't use 8086 with 80287
      .8086
      .287

;Turn privileged mode on and off
      .286P
      .
      .
      .
      .286
```

Ending a Source File

Source files are always terminated with the `END` directive. This directive has two purposes: it marks the end of the source file, and it can indicate the address where execution begins when the program is loaded. The syntax is:

```
END [startaddress]
```

Any statements following the `END` directive are ignored by the assembler. For instance, you can put comments on lines after the `END` directive without using comment specifiers (`;`) or the `COMMENT` directive.

startaddress is a label or expression identifying the address where you want execution to begin when the program is loaded. Specifying a start address is discussed in detail in Section 7.

Section 7

Defining Segment Structure

Segments are a fundamental part of assembly language programming for the 8086 family of processors. They are related to the segmented architecture used by Intel® for its 16 bit and 32 bit microprocessors. Segments are explained in the following subsections:

- Simplified Segment Definitions
- Full Segment Definitions
- Defining Segment Groups
- Associating Segments with Registers
- Initializing Segment Registers
- Nesting Segments

This architecture is explained in more detail in Section 15.

A segment is a collection of instructions or data whose addresses are all relative to the same segment register. Segments can be defined by using simplified segment directives or full segment definitions.

In most cases, simplified segment definitions are a better choice. They are easier to use and more consistent, yet you seldom sacrifice any functionality by using them. Simplified segment directives automatically define the segment structure required when combining assembler modules with modules prepared with Microsoft high level languages.

Although more difficult to use, full segment definitions give more complete control over segments. A few complex programs may require full segment definitions in order to get unusual segment orders and types. In previous versions of MASM, full segment definitions were the only way to define segments, so you may need to use them to maintain existing source code.

This section describes both methods. If you choose to use simplified segment directives, you will probably not need to read about full segment definitions.

Simplified Segment Definitions

The CTOS Microsoft Macro Assembler implements a simplified system for declaring segments. This is explained in the following subsections:

- Understanding Memory Models
- Defining the Memory Model
- Defining Simplified Segments
- Using Predefined Equates
- Simplified Segment Defaults
- Default Segment Names

By default, the simplified segment directives use the segment names and conventions followed by Microsoft high level languages. If you are willing to accept these conventions, the more difficult aspects of segment definition are handled automatically.

If you are writing standalone assembler programs in which segment names, order, and other definition factors are not crucial, the simplified segment directives make programming easier. The Microsoft conventions are flexible enough to work for most kinds of programs. If you are new to assembly language programming, you should use the simplified segment directives for your first programs.

If you are writing assembler routines to be linked with Microsoft high level languages, the simplified segment directives ensure against mistakes that would make your modules incompatible. The names are automatically defined consistently and correctly.

When you use simplified segment directives, ASSUME and GROUP statements that are consistent with Microsoft conventions are generated automatically. You can learn more about the ASSUME and GROUP directives in Defining Segment Groups and Associating Segments with Registers. However, for most programs you do not need to understand these directives. You simply use the simplified segment directives in the format shown in the examples.

Understanding Memory Models

To use simplified segment directives, you must declare a memory model for your program. The memory model specifies the default size of data and code used in a program.

CTOS Microsoft high level languages require that each program have a default size (or memory model). Any assembly language routine called from a high level language program should have the same memory model as the calling program. See the documentation for your language to find out what memory models it can use.

The most commonly used memory models are described in Table 7-1.

Table 7-1. Memory Models

Model	Description
Small	All data fits within a single 64K segment, and all code fits within a 64K segment. Therefore, all code and data can be accessed as near. This is the most common model for standalone assembler programs. C is the only Microsoft language that supports this model.
Medium	All data fits within a single 64K segment, but code may be greater than 64K. Therefore, data is near, but code is far. Most recent versions of Microsoft languages support this model.
Compact	All code fits within a single 64K segment, but the total amount of data may be greater than 64K (although no array can be larger than 64K). Therefore, code is near, but data is far. C is the only Microsoft language that supports this model.
Large	Both code and data may be greater than 64K (although no array can be larger than 64K). Therefore, both code and data are far. All Microsoft languages support this model.
Huge	Both code and data may be greater than 64K. In addition, data arrays may be larger than 64k. Both code and data are far, and pointers to elements within an array must also be far. Most recent versions of Microsoft languages support this model. Segments are the same for large and huge models.

Standalone assembler programs can have any model. Small model is adequate for most programs written entirely in assembly language. Since near data or code can be accessed more quickly, the smallest memory model that can accommodate your code and data is usually the most efficient.

Mixed model programs use the default size for most code and data but override the default for particular data items. Standalone assembler programs can be written as mixed model programs by making specific procedures or variables near or far. Some Microsoft high level languages have NEAR, FAR, and HUGE keywords that enable you to override the default size of individual data or code items.

Defining the Memory Model

The `.MODEL` directive is used to initialize the memory model. This directive should be used early in the source code before any other segment directive. The syntax is:

`.MODEL memorymodel`

The *memorymodel* can be SMALL, MEDIUM, COMPACT, LARGE, or HUGE. Segments are defined the same for large and huge models, but the `@datasize` equate (explained in Using Predefined Equates) is different.

If you are writing an assembler routine for a high level language, *memorymodel* should match the memory model used by the compiler or interpreter.

If you are writing a standalone assembler program, you can use any model. Understanding Memory Models describes each memory model. Small model is the best choice for most standalone assembler programs.

Note: *You must use the .MODEL directive before defining any segment. If one of the other simplified segment directives (such as .CODE or .DATA) is given before the .MODEL directive, an error is generated.*

The statement:

`.MODEL small`

defines default segments for small model programs and creates the ASSUME and GROUP statements used by small model programs. The example statements might be used at the start of the main (or only) module of a standalone assembler program.

The statement:

`.MODEL LARGE`

defines default segments for large model programs and creates the ASSUME and GROUP statements used by large model programs. The example statement might be used at the start of an assembly module that would be called from a large model C, BASIC, FORTRAN, or Pascal program.

Note: *If you use the .386 directive before the .MODEL directive, the segment definitions defines 32 bit segments. If you want to enable the 80386 processor with 16 bit segments, you should give the .386 directive after the .MODEL directive.*

Defining Simplified Segments

The .CODE, .DATA, .DATA?, .FARDATA, .FARDATA?, .CONST, and .STACK directives indicate the start of a segment. They also end any open segment definition used earlier in the source code.

<code>.STACK</code> [<i>size</i>]	Stack segment
<code>.CODE</code> [<i>name</i>]	Code segment
<code>.DATA</code>	Initialized near data segment
<code>.DATA?</code>	Uninitialized near data segment
<code>.FARDATA</code> [<i>name</i>]	Initialized far data segment
<code>.FARDATA?</code> [<i>name</i>]	Uninitialized far data segment
<code>.CONST</code>	Constant data segment

For segments that take an optional name, a default name is used if none is specified. See Default Segment Names for information on default segment names.

Each new segment directive ends the previous segment. The `END` directive closes the last open segment in the source file.

The size argument of the `.STACK` directive is the number of bytes to be declared in the stack. If no size is given, the segment is defined with a default size of one kilobyte.

Standalone assembler programs in the `.RUN` format should define a stack for the main (or only) module. Stacks are defined by the compiler or interpreter for modules linked with a main module from a high level language.

Code should be placed in a segment initialized with the `.CODE` directive, regardless of the memory model. Normally, only one code segment is defined in a source module. If you put multiple code segments in one source file, you must specify name to distinguish the segments. The name can be specified only for models allowing multiple code segments (medium and large). Name is ignored if given with small or compact models.

Uninitialized data is any variable declared by using the indeterminate symbol (?) and the `DUP` operator. When declaring data for modules that will be used with a Microsoft high level language, you should follow the convention of using `.DATA` or `.FARDATA` for initialized data and `.DATA?` or `.FARDATA?` for uninitialized data. For standalone assembler programs, using the `.DATA?` and `.FARDATA?` directives is optional. You can put uninitialized data in any data segment.

Constant data is data that must be declared in a data segment but is not subject to change at run time. Use of this segment is optional for standalone assembler programs. If you are writing assembler routines to be called from a high level language, you can use the `.CONST` directive to declare strings, real numbers, and other constant data that must be allocated as data.

Data in segments defined with the `.STACK`, `.CONST`, `.DATA` or `.DATA?` directives is placed in a group called `DGROUP`. Data in segments defined with the `.FARDATA` or `.FARDATA?` directives is not placed in any group. See *Defining Segment Groups* for more information on segment groups. When initializing the `DS` register to access data in a group associated segment, the value of `DGROUP` should be loaded into `DS`. See *Initializing the DS Register* for information on initializing data segments.

```

                                .MODEL    SMALL
                                .STACK    100h
                                .DATA
ivariable      DB              5
iarray         DW             50 DUP (5)
string         DB             "This is a string"
uarray         DW             50 DUP (?)
                                EXTRN      xvariable:WORD
                                .CODE
start:         mov             ax,DGROUP
                                mov             ds,ax
                                EXTRN      xprocedure:NEAR
                                call          xprocedure
                                .
                                .
                                .
                                END        start

```

This code uses simplified segment directives for a small model, standalone assembler program. Notice that initialized data, uninitialized data, and a string constant are all defined in the same data segment. See Default Segment Names for an equivalent version that uses full segment definitions.

The following example uses simplified segment directives to create a module that might be called from a large model, high level language program.

```

                                .MODEL    LARGE
                                .FARDATA?
fuarray        DW             10 DUP (?)           ; Far uninitialized data
                                .CONST
string         DB             "This is a string"   ; String constant
                                .DATA
niarray        DB             100 DUP (5)          ; Near initialized data
                                .FARDATA
fiarray        EXTR          xvariable:FAR
                                DW             100 DUP (10) ; Far initialized data
                                .CODE
task           EXTR          xprocedure:PROC
                                PROC
                                .
                                .
                                .
                                ret
task           ENDP
                                END

```

Notice that different types of data are put in different segments to conform to Microsoft compiler conventions. See Default Segment Names for an equivalent version using full segment definitions.

Using Predefined Equates

Several equates are predefined for you. You can use the equate names at any point in your code to represent the equate values. You should not assign equates having these names. The predefined equates are listed in Table 7–2

Table 7–2. Predefined Equates

Name	Value
@curseg	<p>This name has the segment name of the current segment. This value may be convenient for ASSUME statements, segment overrides, or other cases in which you need to access the current segment. It can also be used to end a segment, as shown below:</p> <pre>@curseg ENDS ; End current segment .286 ; Must be outside segment .CODE ; Restart segment</pre>
@filename	<p>This value represents the base name of the current source file. For example, if the current source file is task.asm, the value of @filename is task This value can be used in any name you would like to change if the file name changes. For example, it can be used as a procedure name:</p> <pre>@filename PROC . . . @filename ENDP</pre>

continued

Table 7-2. Predefined Equates (cont.)

Name	Value
@codesize and @datasize	<p>If the .MODEL directive has been used, the @codesize value is 0 for small and compact models or 1 for medium, large, and huge models. The @datasize value is 0 for small and medium models, 1 for compact and large models, and 2 for huge models. These values can be used in conditional assembly statements:</p> <pre> IF @datasize les bx,pointer ; Load far pointer mov ax,es:WORD PTR [bx] ELSE mov bx,WORD PTR pointer ; Load near pointer mov ax,WORD PTR [bx] ENDIF </pre>
Segment equate	<p>For each of the primary segment directives, there is a corresponding equate with the same name, except that the equate starts with an at sign (@) but the directive starts with a period. For example, the @code equate represents the segment name defined by the .CODE directive. Similarly, @fardata represents the .FARDATA segment name and @fardata? represents the .FARDATA? segment name. The @data equate represents the group name shared by all the near data segments. It can be used to access the segments created by the .DATA, .DATA?, .CONST, and .STACK segments.</p> <p>These equates can be used in ASSUME statements and at any other time a segment must be referred to by name; for example:</p> <pre> ASSUME es:@fardata ; Assume ES to far data ; (.MODEL handles DS) mov ax,@data ; Initialize near to DS mov ds,ax mov ax,@fardata ; Initialize far to ES mov es,ax </pre>

Note: Although predefined equates are part of the simplified segment system, the @curseg and @filename equates are also available when using full segment definitions.

Simplified Segment Defaults

When you use the simplified segment directives, defaults are different in certain situations than they would be if you gave full segment definitions. Defaults that change are listed below:

- If you give full segment definitions, the default size for the PROC directive is always NEAR. If you use the .MODEL directive, the PROC directive is associated with the specified memory model: NEAR for small and compact models and FAR for medium, large, and huge models. See Section 8 for further discussion of the PROC directive.
- If you give full segment definitions, the segment address used as the base when calculating an offset with the OFFSET operator is the data segment (the segment associated with the DS register). With the simplified segment directives, the base address is the DGROUP segment for segments that are associated with a group. This includes segments declared with the .DATA, .DATA?, and .STACK directives, but not segments declared with the .CODE, .FARDATA, and .FARDATA? directives.

For example, assume the variable test1 was declared in a segment defined with the .DATA directive and test2 was declared in a segment defined with the .FARDATA directive. The statement:

```
mov    ax,OFFSET test1
```

loads the address of test1 relative to DGROUP. The statement

```
mov    ax,OFFSET test2
```

loads the address of test2 relative to the segment defined by the .FARDATA directive. See Defining Segment Groups for more information on groups.

Default Segment Names

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, it is possible to mix full segment definitions with simplified segment definitions. Therefore, programmers may want to know the actual names assigned to all segments.

Table 7–3 shows the default segment names created by each directive.

Table 7–3. Default Segments and Types for Standard Memory Models

Model	Directive	Name	Align	Combine	Class	Group
Small	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Medium	.CODE	<i>name</i> _TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Compact	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP

continued

Table 7–3. Default Segments and Types for Standard Memory Models (cont.)

Model	Directive	Name	Align	Combine	Class	Group
Large or huge	.CODE	<i>name</i> _TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	· PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	· PARA	STACK	'STACK'	DGROUP

The *name* used as part of far code segment names is the file name of the module. The default name associated with the .CODE directive can be overridden in medium and large models. The default names for the .FARDATA and .FARDATA? directives can always be overridden.

The segment and group table at the end of listings always shows the actual segment names. However, the group and assume statements generated by the .MODEL directive are not shown in listing files. For a program that uses all possible segments, group statements equivalent to the following are generated:

```
DGROUP      GROUP    _DATA,CONST,_BSS,STACK
```

For small and compact models, the following is generated:

```
ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
```

For medium, large, and huge models the following statement is given:

```
ASSUME  cs:name_TEXT,ds:DGROUP,ss:DGROUP
```

Note: If the .386 directive is used, the default align type for all segments is *DWORD*.

The following example is equivalent to the first example in Defining Simplified Segments:

```

                                EXTRN    xvariable:WORD
                                EXTRN    xprocedure:NEAR
DGROUP                        GROUP    _DATA,_BSS
                                ASSUME    cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT                        SEGMENT WORD PUBLIC 'CODE'
start:                        mov     ax,DGROUP
                                mov     ds,ax
                                .
                                .
                                .
                                ENDS
_TEXT                        SEGMENT WORD PUBLIC 'DATA'
_DATA                        DB        5
ivariable                    DW        50 DUP (5)
iarray                       DB        "This is a string"
string                       DW        50 DUP (?)
uarray                       DB
_DATA                        ENDS
STACK                        SEGMENT PARA STACK 'STACK'
                                DB        100h DUP (?)
STACK                        ENDS
                                END      start

```

The external variables are declared at the start of the source code in this example. With simplified segment directives, they can be declared in the segment in which they are used.

This following example is equivalent to the second example in Defining Simplified Segments.

```

DGROUP                        GROUP    _DATA,CONST,STACK
                                ASSUME    cs:TASK_TEXT,ds:FAR_DATA,ss:STACK
                                EXTRN    xprocedure:FAR
                                EXTR      xvariable:FAR
FAR_BSS                      SEGMENT PARA 'FAR_DATA'
fuarray                      DW        10 DUP (?)           ; Far uninitialized data
FAR_BSS                      ENDS
CONST                        SEGMENT WORD PUBLIC 'CONST'
string                      DB        "This is a string"    ; String constant
CONST                        ENDS
_DATA                        SEGMENT WORD PUBLIC 'DATA'
niarray                      DB        100 DUP (5)          ; Near initialized data
_DATA                        ENDS
FAR_DATA                     SEGMENT WORD 'FAR_DATA'
fiarray                      DW        100 DUP (10)
FAR_DATA                     ENDS

```

```
TASK_TEXT      SEGMENT WORD PUBLIC 'CODE'
task           PROC      FAR
               .
               .
               .
               ret
task           ENDP
TASK_TEXT      ENDS
               END
```

Notice that the segment order is the same in both versions. The segment order shown here is written to the object file, but it is different in the executable file. The segment order specified by the compiler overrides the segment order in the module object file.

Full Segment Definitions

If you need complete control over segments, you may want to give complete segment definitions. The following subsections explain all aspects of segment definitions, including how to order segments and how to define all the segment types.

Setting the Segment Order Method

The order in which MASM writes segments to the object file can be either sequential or alphabetical. If the sequential method is specified, segments are written in the order in which they appear in the source code. If the alphabetical method is specified, segments are written in the alphabetical order of their segment names.

The default is sequential. If no segment order directive or option is given, segments are ordered sequentially. The segment order method is only one factor in determining the final order of segments in memory. The `DOSSEG` directive (see [Specifying DOS Segment Order](#)) and class type (see [Controlling Segment Structure with Class Type](#)) can also affect segment order.

The ordering method can be set by using the `.ALPHA` or be set using the `/S` (sequential) or `/A` (alphabetical) assembler options (see [Section 4](#)). The directives have precedence over the options. For example, if the source code contains the `.ALPHA` directive, but the `/S` option is given on the command line, the segments are ordered alphabetically.

Changing the segment order is an advanced technique. In most cases, you can simply leave the default sequential order in effect. If you are linking with high level language modules, the compiler automatically sets the segment order.

Some previous versions of the IBM Macro Assembler ordered segments alphabetically by default. If you have trouble assembling and linking source code listings from books or magazines, try using the /A option. Listings written for previous IBM versions of the assembler may not work without this option.

In the first example, the DATA segment is written to the object file first because it appears first in the source code. In the second example, the CODE segment is written to the object file first because its name comes first alphabetically.

```
DATA      .SEQ
DATA      SEGMENT WORD PUBLIC 'DATA'
DATA      ENDS
CODE      SEGMENT WORD PUBLIC 'CODE'
CODE      ENDS
```

```
DATA      .ALPHA
DATA      SEGMENT WORD PUBLIC 'DATA'
DATA      ENDS
CODE      SEGMENT WORD PUBLIC 'CODE'
CODE      ENDS
```


Defining Full Segments

The beginning of a program segment is defined with the `SEGMENT` directive, and the end of the segment is defined with the `ENDS` directive. The syntax is:

```
name SEGMENT [ align ] [ combine ] [ use ] [ 'class' ]  
statements  
ENDS
```

name defines the name of the segment. This name can be unique or it can be the same name given to other segments in the program. Segments with identical names are treated as the same segment. For example, if it is convenient to put different portions of a single segment in different source modules, the segment is given the same name in both modules.

The optional *align*, *combine*, *use*, and 'class' types give the linker and the assembler instructions on how to set up and combine segments. Types can be specified in any order, it is not necessary to enter all types, or any type, for a given segment.

Defining segment types is an advanced technique. Beginning assembly language programmers might try using the simplified segment directives discussed in Simplified Segment Definitions.

Do not confuse the `PAGE` align type and the `PUBLIC` combine type with the `PAGE` and `PUBLIC` directives. The distinction should be clear from context since the align and combine types are used only on the same line as the `SEGMENT` directive.

Controlling Alignment with Align Type

The optional align type defines the range of memory addresses from which a starting address for the segment can be selected. The align types, as show in Table 7–4, can be any one of the following:

Table 7–4. Align Types

Align Type	Meaning
BYTE	Uses the next available byte address.
WORD	Uses the next available word address (2 bytes per word).
DWORD	Uses the next available doubleword address (4 bytes per doubleword); the DWORD align type is normally used in 32 bit segments with the 80386.
PARA	Uses the next available paragraph address (16 bytes per paragraph).
PAGE	Uses the next available page address (256 bytes per page).

If no align type is given, PARA is used by default.

The linker uses the alignment information to determine the relative start address for each segment.

Align types are illustrated in Figure 7–1.

Setting Segment Word Size with Use Type

The *use* type specifies the segment word size on the 80386 processor. Segment word size is the default operand and address *size of a segment*.

The *use* type can be USE16 or USE32. These types are relevant only if you have enabled 80386 instructions and addressing modes with the .386 directive. The assembler generates an error if you specify *use type* when the 80386 processor is not enabled.

With the 80286 and other 16 bit processors, the segment word size is always 16 bits. A 16 bit segment can contain up to 65,536 (64K) bytes. However, the 80386 is capable of using either 16 bit or 32 bit segments. A 32 bit segment can contain up to 4,294,967,296 bytes (4 gigabytes). Although MASM permits you to define 4 gigabyte segments in 32 bit segments.

If you do not specify a use type, the segment word size is 32 bits by default when the .386 directive is used.

The effect of addressing modes is changed by the word size you specify for the code segment. See Section 16 for more information on 80386 addressing modes. The meaning of the WORD and DWORD type specifiers is not changed by the use type. WORD always indicates 16 bits and DWORD always indicates 32 bits regardless of the current segment word size.

Although the assembler allows you to use 16 bit and 32 bit segments in the same program, you should normally make all segments the same size. Mixing segment sizes is an advanced technique that can have unexpected side effects. For the most part, it is used only by systems programmers.

```
; 16 bit segment
        .386
_DATA    SEGMENT DWORD USE16 PUBLIC 'DATA'
        .
        .
_DATA    ENDS

; 32 bit segment
_TEXT    SEGMENT DWORD USE32 PUBLIC 'CODE'
        .
        .
_TEXT    ENDS
```

Defining Segment Combinations with Combine Type

The optional combine type defines how to combine segments having the same name. The combine types, as described in Table 7–5, can be any one of the following:

Table 7–5. Combine Types

Combine Type	Meaning
PUBLIC	Concatenates all segments having the same name to form a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the segment.
STACK	Concatenates all segments having the same name to form a single, contiguous segment. This combine type is the same as the PUBLIC combine type, except that all addresses in the new segment are relative to the SS segment register. The stack pointer (SP) register is initialized to the length of the segment. The stack segment of your program should normally use the STACK type, since this automatically initializes the SS register, as described in Initializing the SS and SP Registers. If you create a stack segment and do not use the STACK type, you must give instructions to initialize the SS and SP registers.
COMMON	Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address. If variables are initialized in more than one segment having the same name and COMMON type, the most recently initialized data replace any previously initialized data.
MEMORY	Concatenates all segments having the same name to form a single, contiguous segment.

continued

Table 7-5. Combine Types (cont.)

Combine Type	Meaning
AT address	<p>Causes all label and variable addresses defined in the segment to be relative to address.</p> <p>The address can be any valid expression, but must not contain a forward reference—that is, a reference to a symbol defined later in the source file. An AT segment typically contains no code or initialized data. Instead, it represents an address template that can be placed over code or data already in memory, such as a screen buffer or other absolute memory locations defined by hardware. The linker will not generate any code or data for AT segments, but existing code or data can be accessed by name if it is given a label in an AT segment. Section 8 shows an example of a segment with AT combine type.</p> <p>The AT combine type has no meaning in protected mode programs, since the segment represents a movable selector rather than a physical address. Real mode programs that use AT segments must be modified before they can be used in protected mode.</p>

If no combine type is given, the segment has private type. Segments having the same name are not combined. Instead, each segment receives its own physical segment when loaded into memory.

Note: *Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict. If types are given for an initial segment definition, then subsequent definitions for that segment need not specify any types.*

Normally you should provide at least one stack segment (having STACK combine type) in a program. If no stack segment is declared, LINK displays a warning message. You can ignore this message if you have a specific reason for not declaring a stack segment.

The following source code shell illustrates one way in which the combine and align types can be used. Figure 7–1 shows the way LINK would load the sample program into memory.

```

NAME module_1

ASEG      SEGMENT WORD PUBLIC 'CODE'
start:    .
          .
          .
ASEG      ENDS
BSEG      SEGMENT WORD COMMON 'DATA'
          .
          .
          .
BSEG      ENDS
CSEG      SEGMENT PARA STACK 'STACK'
          .
          .
          .
CSEG      ENDS
DSEG      SEGMENT AT 0B800H
          .
          .
          .
DSEG      ENDS
          END start

NAME module_2

ASEG      SEGMENT WORD PUBLIC 'CODE'
          .
          .
          .
ASEG      ENDS
BSEG      SEGMENT WORD COMMON 'DATA'
          .
          .
          .
BSEG      ENDS

```

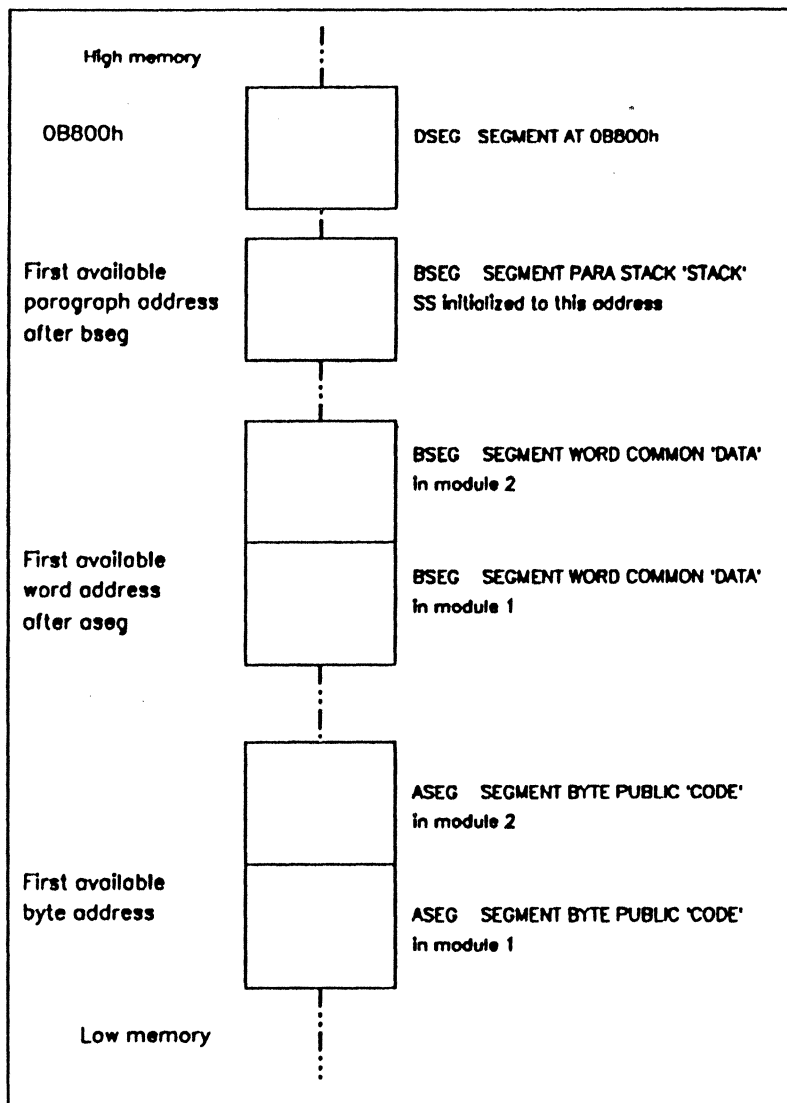


Figure 7-1. Segment Structure with Combine and Align Types

Controlling Segment Structure with Class Type

Class type is a means of associating segments that have different names, but similar purposes. It can be used to control segment order and to identify the code segment.

The class name must be enclosed in single quotation marks ('). Class names are not case sensitive unless the /ML or /MX option is used during assembly.

All segments belong to a class. Segments for which no class name is explicitly stated have the null class name. LINK imposes no restriction on the number or size of segments in a class. The total size of all segments in a class can exceed 64K.

Note: *The names assigned for class types of segments should not be used for other symbol definitions in the source file. For example, if you give a segment the class name 'CONSTANT', you should not give the name constant to variables or labels in the source file.*

The linker expects segments having the class name CODE or a class name with the suffix CODE to contain program code. You should always assign this class name to segments containing code.

Class type is one of two factors that control the final order of segments in an executable file. The other factor is the order of the segments in the source file (with the /S option or the (with the /A option or the .ALPHA directive).

These factors control different internal behavior, but both affect final order of segments in the executable file. The sequential or alphabetical order of segments in the source file determines the order in which the assembler writes segments to the object file. The class type can affect the order in which the linker writes segments from object files to the executable file.

Segments having the same class type are loaded into memory together, regardless of their sequential or alphabetical order in the source file.

When MASM assembles the following program fragment, it writes the segments to the object file in sequential or alphabetical order, depending on whether the /A option or the .ALPHA directive was used. The sequential and alphabetical order are the same, so the order will be A_SEG, B_SEG, C_SEG in either case.

```
A_SEG  SEGMENT  'SEG_1'
A_SEG  ENDS

B_SEG  SEGMENT  'SEG_2'
B_SEG  ENDS

C_SEG  SEGMENT  'SEG_1'
C_SEG  ENDS
```

When the linker writes the segments to the executable file, it first checks to see if any segments have the same class type. If they do, it writes them to the executable file together. Thus A_SEG and C_SEG are placed together because they both have class type 'SEG_1'. The final order in memory is A_SEG, C_SEG, B_SEG.

Since LINK processes modules in the order it receives them on the command line, you may not always be able to easily specify the order you want segments to be loaded. For example, assume your program has four segments that you want loaded in the following order: _TEXT, _DATA, CONST, and STACK.

The _TEXT, CONST, and STACK segments are defined in the first module of your program, but the _DATA segment is defined in the second module. LINK will not put the segments in the proper order because it first loads the segments encountered in the first module.

You can avoid this problem by starting your program with dummy segment definitions in the order you wish to load your real segments. The dummy segments can either go at the start of the first module, or they can be placed in a separate include file that is called at the start of the first module. You can then put the actual segment definitions in any order or any module you find convenient.

For example, you might call the following include file at the start of the first module of your program:

```
_TEXT      SEGMENT WORD PUBLIC 'CODE'
_TEXT      ENDS
_DATA      SEGMENT WORD PUBLIC 'DATA'
_DATA      ENDS
CONST      SEGMENT WORD PUBLIC 'CONST'
CONST      ENDS
STACK      SEGMENT PARA STACK 'STACK'
STACK      ENDS
```

Once a segment has been defined, you do not need to specify the align, combine, use, and class types on subsequent definitions. For example, if your code defined dummy segments as shown above, you could define an actual data segment with the following statements:

```
_DATA      SEGMENT
            .
            .
            .
_DATA      ENDS
```

Defining Segment Groups

A group is a collection of segments associated with the same starting address. You may wish to use a group if you want several types of data to be organized in separate segments in your source code, but want them all to be accessible from a single, common segment register at run time. The syntax is:

name *GROUP* segment [,segment]..

name is the symbol assigned to the starting address of the group. All labels and variables defined within the segments of the group are relative to the start of the group, rather than to the start of the segments in which they are defined.

segment can be any previously defined segment or a SEG expression (see Section 11).

Segments can be added to a group one at a time. For example, you can define and add segments to a group one by one. This is a new feature. Previous versions required that all segments in a group be defined at one time.

The GROUP directive does not affect the order in which segments of a group are loaded. Loading order depends on each segment's class, or on the order in which object modules are given to the linker.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65,535 bytes.

Note: *When the MODEL directive is used, the offset of a group relative segment refers to the ending address of the segment, not the beginning. For example, the expression OFFSET STACK evaluates to the end of the stack segment.*

Group names can be used with the ASSUME directive (discussed in Associating Segments with Registers) and as an operand prefix with the segment override operator (covered in Section 11).

```
DGROUP      GROUP  ASEG,CSEG
            ASSUME ds:DGROUP

ASEG        SEGMENT WORD PUBLIC 'DATA'
            .
asym        .
            .
ASEG        ENDS

BSEG        SEGMENT WORD PUBLIC 'DATA'
            .
bsym        .
            .
BSEG        ENDS

CSEG        SEGMENT WORD PUBLIC 'DATA'
            .
csym        .
            .
CSEG        ENDS
            END
```

Figure 7–2 shows the order of the example segments in memory. They are loaded in the order in which they appear in the source code (or in alphabetical order if the .ALPHA directive or /A option is specified).

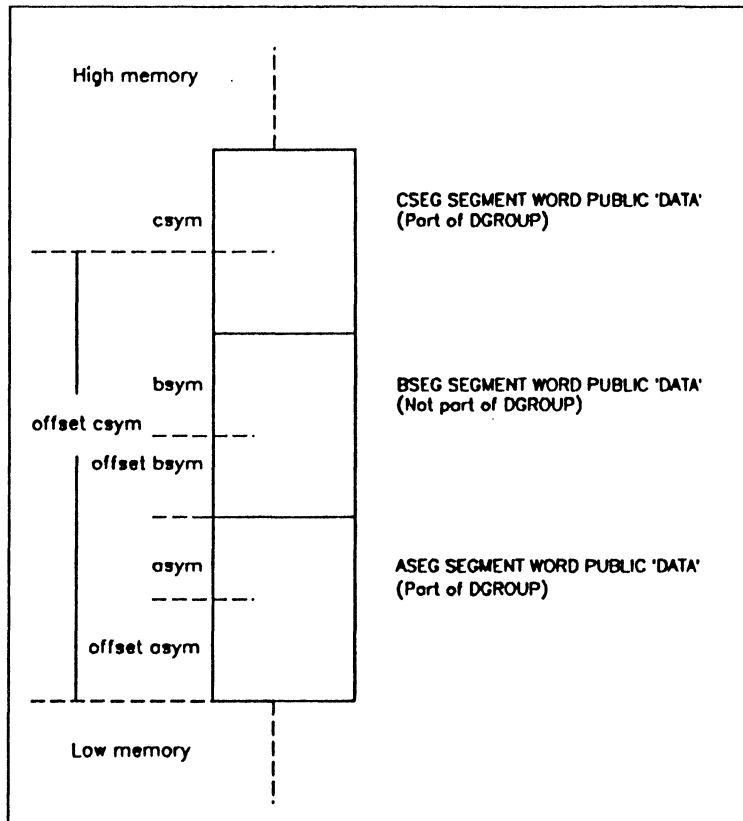


Figure 7-2. Segment Structure with Groups

Since ASEG and CSEG are declared part of the same group, they have the same base despite their separation in memory. This means that the symbols `asym` and `csym` have offsets from the beginning of the group, which is also the beginning of ASEG. The offset of `bsym` is from the beginning of BSEG, since it is not part of the group. This sample illustrates the way LINK organizes segments in a group. It is not intended as a typical use of a group.

Associating Segments with Registers

Many instructions assume a default segment. For example, JMP instructions assume the segment associated with the CS register; PUSH and POP instructions assume the segment associated with the SS register; MOV instructions assume the segment associated with the DS register.

When the assembler needs to reference an address, it must know what segment the address is in. It does this by using default segment or group addresses assigned with the ASSUME directive.

Note: *Using the ASSUME directive to tell the assembler which segment to associate with a segment register is not the same as telling the processor. The ASSUME directive only affects assembly time assumptions. You may need to use instructions to change run time assumptions. Initializing segment registers at run time is discussed in Initializing Segment Registers. The syntax for the ASSUME directive is:*

```
ASSUME segmentregister:name [ ,segmentregister:name ]...  
ASSUME segmentregister:NOTHING  
ASSUME NOTHING
```

name must be the name of the segment or group that is to be associated with *segmentregister*. Subsequent instructions that assume a default register for referencing labels or variables automatically assume that if the default segment is *segmentregister*, then the label or variable is in the *name* segment or group.

The ASSUME directive can define a segment for each of the segment registers. The *segmentregister* can be CS, DS, ES, or SS (FS and GS are also available on the 80386). *name* must be one of the following:

- The name of a segment defined in the source file with the SEGMENT directive
- The name of a group defined in the source file with the GROUP directive
- The keyword NOTHING
- A SEG expression (see Section 11)
- A string equate that evaluates to a segment or group name (but not a string equate that evaluates to a SEG expression)

The keyword **NOTHING** cancels the current segment selection. For example, the statement **ASSUME NOTHING** cancels all register selections made by previous **ASSUME** statements.

Usually a single **ASSUME** statement defines all four segment registers at the start of the source file. However, you can use the **ASSUME** directive at any point to change segment assumptions.

Using the **ASSUME** directive to change segment assumptions is often equivalent to changing assumptions with the segment override operator (:) (see Section 11). The segment override operator is more convenient for one time overrides, whereas the **ASSUME** directive may be more convenient if previous assumptions must be overridden for a sequence of instructions.

```

                                .MODEL  large      ; DS automatically assumed to @data
                                .STACK  100h
                                .DATA
d1                               DW      7
                                .FARDATA
d2                               DW      9

                                .CODE
start:                           mov     ax,@data    ; Initialize near data
                                mov     ds,ax
                                mov     ax,@fardata   ; Initialize far data
                                mov     es,ax
                                .
                                .
                                .
; Method 1 for series of instructions that need override
; Use segment override for each statement
                                mov     ax,es:d2
                                .
                                .
                                mov     es:d2,bx
; Method 2 for series of instructions that need override
; Use ASSUME at beginning of series of instructions
                                ASSUME es:@fardata
                                mov     cx,d2
                                .
                                .
                                .
                                mov     d2,dx

```

Initializing Segment Registers

Assembly language programs must initialize segment values for each segment register before instructions that reference the segment register can be used in the source program.

Initializing segment registers is different from assigning default values for segment registers with the ASSUME statement. The ASSUME directive tells the assembler what segments to use at assembly time. Initializing segments gives them an initial value that will be used at run time.

Each of the segment registers is initialized in a different way as explained in the following subsections:

- Initializing the CS and IP Registers
- Initializing the SS and SP Registers
- Initializing the ES Register

Initializing the CS and IP Registers

The CS and IP registers are initialized by specifying a starting address with the END directive. The syntax is:

END [*startaddress*]

startaddress is a label or expression identifying the address where you want execution to begin when the program is loaded. Normally a label for the *startaddress* should be placed at the address of the first instruction in the code segment.

The CS segment is initialized to the value of *startaddress*. The IP register is normally initialized to 0. You can change the initial value of the IP register by using the ORG directive (see Section 8) just before the *startaddress* label.

If a program consists of a single source module, then *startaddress* is required for that module. If a program has several modules, all modules must terminate with an **END** directive, but only one of them can define a start address.

WARNING

One, and only one, module must define a *startaddress*. If you do not specify *startaddress*, none is assumed. Neither MASM nor LINK will generate an error message, but your program will probably start execution at the wrong address.

In the following examples, if Module 1 and Module 2 are linked into a single program, it is essential that only the calling module define a starting address.

```
; Module 1
start:      .CODE                      ; First executable instruction
            .
            .
            EXTRN    task:NEAR
            call     task
            .
            .
            END      start            ; Starting address defined in main mod

; Module 2
PUBLIC      task
.CODE
task       PROC
            .
            .
task       ENDP
            END                      ; No starting address in secondary module
```


Initializing the DS Register

The DS register must be initialized to the address of the segment that will be used for data.

The address of the segment or group for the initial data segment must be loaded into the DS register. This is done in two statements because a memory value cannot be loaded directly into a segment register. The segment setup lines typically appear at the start or very near the start of the code segment.

```
_DATA          SEGMENT WORD PUBLIC 'DATA'
               .
               .
               .
_DATA          ENDS
_TEXT          SEGMENT BYTE PUBLIC 'CODE'
start:         ASSUME  cs:_TEXT,ds:_DATA
               mov     ax,_DATA          ; Load start of data segment
               mov     ds,ax            ; Transfer to DS register
               .
               .
               .
_TEXT          ENDS
               END      start
```

If you are using the Microsoft naming convention and segment order, the address loaded into the DS register is not a segment address but the address of DGROUP, as shown in the example below. With simplified segment directives, the address of DGROUP is represented by the predefined equate @data

```
               .MODEL  SMALL
               .DATA
               .
               .
               .
start:         .CODE
               mov     ax,@data          ; Load start of DGROUP (@data)
               mov     ds,ax            ; Transfer to DS register
               .
               .
               .
               END      start
```

Initializing the SS and SP Registers

The SS register is automatically initialized to the value of the last segment in the source code having combine type STACK. The SP register is automatically initialized to the size of the stack segment. Thus SS:SP initially points to the end of the stack.

If you use a stack segment with combine type STACK, initialization of SS and SP is automatic. The stack is automatically set up in this way with the simplified segment directives.

However, you can initialize or reinitialize the stack segment directly by changing the values of SS and SP. Since hardware interrupts use the same stack as the program, you should turn off hardware interrupts while changing the stack. Most 8086 family processors do this automatically, but early versions of the 8088 do not.

The following example reinitializes SS so that it has the same value as DS, and adjusts SP to reflect the new stack offset. Microsoft high level language compilers do this so that stack variables in near procedures can be accessed relative to either SS or DS.

```

.MODEL    small
.STACK   100h           ; Initialize STACK
.DATA
.
.
.
.CODE
start:   mov     ax,@data      ; Load segment location
         mov     ds,ax         ; into DS register
         cli                     ; Turn off interrupts
         mov     ss,ax         ; Load same value as DS into SS
         mov     sp,OFFSET STACK ; Give SP new stack size
         sti                     ; Turn interrupts back on
.
.
.
```

Initializing the ES Register

The ES register is not automatically initialized. If your program uses the ES register, you must initialize it by moving the appropriate segment value into the register. As follows:

```
ASSUME es:@fardata ; Tell the assembler
mov ax,@fardata ; Tell the processor
mov es,ax
```

Nesting Segments

Segments can be nested. When MASM encounters a nested segment, it temporarily suspends assembly of the enclosing segment and begins assembly of the nested segment. When the nested segment has been assembled, MASM continues assembly of the enclosing segment.

Nesting of segments makes it possible to mix segment definitions in programs that use simplified segment directives for most segment definitions. When a full segment definition is given, the new segment is nested in the simplified segment in which it is defined.

In the following example, a macro called from inside of the code segment (`_TEXT`) allocates a variable within a nested data segment (`_DATA`). This has the effect of allocating more data space on the end of the data segment each time the macro is called. The macro can be used for messages appearing only once in the source code.

```
; Macro to print message on the screen
; Uses full segment definitions – segments nested
```

```
message MACRO text
LOCAL symbol
_DATA SEGMENT WORD PUBLIC 'DATA'
symbol DB &text
DB 13,10,"$"
_DATA ENDS
push ds
mov dx, OFFSET symbol
push dx
call zprint
ENDM

_TEXT SEGMENT BYTE PUBLIC 'CODE'
.
.
.
message "Please insert disk"
```

Although the following example has the same practical effect as the previous, MASM handles the two macros differently. In the first example, assembly of the outer (code) segment is suspended rather than terminated. In the second example, assembly of the code segment terminates, assembly of the data segment starts and terminates, and then assembly of the code segment is restarted.

```
; Macro to print message on the screen
; Uses simplified segment directives – segments not nested

message      MACRO  text
              LOCAL  symbol
              .DATA
symbol        DB      &text
              DB      13,10,"$"
              .CODE
              push    ds
              mov     dx, OFFSET symbol
              push    dx
              call    zprint
              ENDM

              .CODE
              .
              .
              .
              message "Please insert disk"
```


Section 8

Defining Labels and Variables

This section explains how to define labels, variables, and other symbols that refer to instruction and data locations within segments. It contains the following subsections:

- Using Type Specifiers
- Defining Code Levels
- Defining and Initializing Data
- Setting the Location Counter
- Aligning Data

The label and variable definition directives described in this section are closely related to the segment definition directives covered in Section 7. Segment directives assign the addresses for segments. The variable and label definition directives assign offset addresses within segments.

The assembler assigns offset addresses for each segment by keeping track of a value called the location counter. The location counter is incremented as each source statement is processed so that it always contains the offset of the location being assembled. When a label or a variable name is encountered, the current value of the location counter is assigned to the symbol.

This section tells you how to assign labels and most kinds of variables. (Multifield variables such as structures and records are discussed in Section 9.) The section also discusses related directives, including those that control the location counter directly.

Using Type Specifiers

Some statements require type specifiers to give the size or type of an operand. There are two kinds of type specifiers: those that specify the size of a variable or other memory operand, and those that specify the distance of a label.

The type specifiers that give the size of a memory operand are listed in 8–1 with the number of bytes specified by each.

Table 8–1. Type Specifiers For Memory Operand Sizes

Specifier	Number of Bytes
BYTE	1
WORD	2
DWORD	4
FWORD	6
QWORD	8
TBYTE	10

In some contexts, ABS can also be used as a type specifier that indicates an operand is a constant rather than a memory operand. The type specifiers that give the distance of a label are listed in Table 8–2.

Table 8–2. Type Specifiers For Constants

Specifier	Description
FAR	The label references both the segment and offset of the label.the offset of the label.
PROC	The label has the default type (near or far) of the current memory model. The default size is always near if you use full segment definitions. If you use simplified segment definitions (see Section 7) the default type is near for small and compact models or far for medium, large, and huge models.

Directives that use type specifiers include LABEL, PROC, EXTRN, and COMM. Operators that use type specifiers include PTR and THIS.

Defining Code Labels

Code labels give symbolic names to the addresses of instructions in the code segment. These labels can be used as the operands to jump, call, and loop instructions to transfer program control to a new instruction. There are three types of code labels: near labels, procedure labels, and labels created with the LABEL directive. They are explained in the following subsections:

- Near code labels
- Procedure labels
- Code labels defined with the LABEL directive

Near Code Labels

Near label definitions create instruction labels that have NEAR type. These instruction labels can be used to access the address of the label from other statements. The syntax is:

name:

name must not be previously defined in the module and it must be followed by a colon (:). Furthermore, the segment containing the definition must be the one that the assembler currently associates with the CS register. The ASSUME directive is used to associate a segment with a segment register (see Section 7.)

A near label can appear on a line by itself or on a line with an instruction. The same label name can be used in different modules as long as each label is referenced only by instructions in its own module. If a label must be referenced by instructions in another module, it must be given a unique name and declared with the `PUBLIC` and `EXTRN` directives, as described in Section 10. For example:

```
        cmp     ax,5           ; Compare with 5
        ja      bigger
        jb      smaller
        .
        .                     ; Instructions if AX = 5
        .
bigger:  jmp     done           ; Instructions if AX > 5
        .
        .
smaller: jmp     done           ; Instructions if AX < 5
        .
        .
done:
```

Procedure Labels

The start of an assembly language procedure can be defined with the `PROC` directive, and the end of the procedure can be defined with the `ENDP` directive. The syntax is

```
label PROC [[NEAR|FAR]]
statements
RET [[constant]]
label ENDP
```

label assigns a symbol to the procedure. The distance can be `NEAR` or `FAR`. Any `RET` instructions within the procedure automatically have the same distance (`NEAR` or `FAR`) as the procedure. Procedures and the `RET` instruction are discussed in more detail in *Using Procedures*.

The `ENDP` directive labels the address where the procedure ends. Every procedure label must have a matching `ENDP` label to mark the end of the procedure. MASM generates an error message if it does not find an `ENDP` directive to match each `PROC` directive.

When the PROC label definition is encountered, the assembler sets the label's value to the current value of the location counter and sets its type to NEAR or FAR. If the label has FAR type, the assembler also sets its segment value to that of the enclosing segment. If you have specified full segment definitions, the default distance is NEAR. If you are using simplified segment definitions, the default distance is the distance associated with the declared memory model—that is, NEAR for small and compact models or FAR for medium, large, and huge models.

The procedure label can be used in a CALL instruction to direct execution control to the first instruction of the procedure. Control can be transferred to a NEAR procedure label from any address in the same segment as the label. Control can be transferred to a FAR procedure label from an address in any segment.

Procedure labels must be declared with the PUBLIC and EXTRN directives if they are located in one module but called from another module, as shown the following example.

```

                call    task        ; Call procedure
                .
                .
task PROC      NEAR        ; Start of procedure
                .
                .
                ret
task ENDP                      ; End of procedure
```

Section 10 describes using multiple modules.

Code Labels Defined with the LABEL Directive

The LABEL directive provides an alternative method of defining code labels. The syntax is:

name LABEL distance

name is the symbol name assigned to the label. *distance* can be a type specifier such as NEAR, FAR, or PROC. PROC means NEAR or FAR, depending on the default memory model, as described in Section 6. You can use the LABEL directive to define a second entry point into a procedure. FAR code labels can also be the destination of far jumps or of far calls that use the RETF instruction for example:

```
task    PROC        FAR        ; Main entry point
        .
        .
task1   LABEL        FAR        ; Secondary entry point
        .
        .
        ret
task    ENDP          ; End of procedure
```

Defining and Initializing Data

The data definition directives enable you to allocate memory for data. At the same time, you can specify the initial values for the allocated data. Data can be specified as numbers, strings, or expressions that evaluate to constants. The assembler translates these constant values into binary bytes, words, or other units of data. The encoded data are written to the object file at assembly time. The different data types are as follows:

- Variables
- Arrays and buffers
- Variables with labels

Variables

Variables consist of one or more named data objects of a specified size. The syntax is:

`[[name]] directive initializer [[initializer]]..`

name is the symbol name assigned to the variable. If no name is assigned, the data is allocated; but the starting address of the variable has no symbolic name.

The size of the variable is determined by directive. The directives that can be used to define single item data objects are listed in Table 8–3.

Table 8–3. Assembly Directives

Directive	Meaning
DB	Defines byte
DW	Defines word (2 bytes)
DD	Defines doubleword (4 bytes)
DF	Defines farword (6 bytes); normally used only with 80386 processor
DQ	Defines quadword (8 bytes)
DT	Defines 10 byte variable

The optional initializer can be a constant, an expression that evaluates to a constant, or a question mark (?). The question mark is the symbol indicating that the value of the variable is undefined. You can define multiple values by using multiple initializers separated by commas, or by using the DUP operator, as explained in Arrays and Buffers.

Simple data types can allocate memory for integers, strings, addresses, or real numbers.

Integer Variables

When defining an integer variable, you can specify an initial value as an integer constant or as a constant expression. MASM generates an error if you specify an initial value too large for the specified variable.

Integer values for all sizes except 10 byte variables are stored in binary form. They can be interpreted as either signed or unsigned numbers. For instance, the hexadecimal value 0FFCD can be interpreted either as the signed number –51 or the unsigned number 65,485.

The processor cannot tell the difference between signed and unsigned numbers. Some instructions are designed specifically for signed numbers. It is the programmer's responsibility to decide whether a value is to be interpreted as signed or unsigned, and then to use the appropriate instructions to handle the value correctly.

The directives for defining integer variables are listed in table 8–4 with the sizes of integer they can define:

Table 8–4. Directive Descriptions

Directive	Size
DB (bytes)	Allocates unsigned numbers from 0 to 255 or signed numbers from –128 to 127. These values can be used directly in 8086 family instructions.
DW (words)	<p>Allocates unsigned numbers from 0 to 65,535 or signed numbers from –32,768 to 32,767. The bytes of a word integer are stored in the format shown in Figure 8–1:</p> <p>Note that in assembler listings and in most debuggers the bytes of a word are shown in the opposite order—high byte first—since this is the way most people think of numbers. (See Figure 8–1.)</p> <p>Word values can be used directly in 8086 family instructions. They can also be loaded, used in calculations, and stored with 8087 family instructions.</p>
DD (doublewords)	<p>Allocates unsigned numbers from 0 to 4,294,967,295 or signed numbers from –2,147,483,648 to 2,147,483,647. The words of a doubleword integer are stored in the format shown in Figure 8–2.</p> <p>These 32 bit values (called long integers) can be loaded, used in calculations, and stored with 8087 family instructions. Some calculations can be done on these numbers directly with 16 bit 8086 family processors; others involve an indirect method of doing calculations on each word separately (see Section 18). These long integers can be used directly in calculations with the 80386 processor.</p>
DF (farwords)	<p>Allocates 6 byte (48 bit) integers.</p> <p>These values are normally only used as pointer variables on the 80386 processor (see Pointer Variables).</p>

continued

Table 8–4. Directive Descriptions (cont.)

Directive	Size
DQ (quadwords)	<p>Allocates 64 bit integers. The doublewords of a quadword integer are stored in the format shown in Figure 8–3.</p> <p>These values can be loaded, used in calculations, and stored with 8087 family instructions. You must write your own routines to use them with 16 bit 8086 family processors. Some calculations can be done on these numbers directly with the 80386 processor, but others require an indirect method of doing calculations on each doubleword separately (see Section 18).</p> <p>Allocates 10 byte (80 bit) integers if the D radix specifier is used.</p> <p>By default, DT allocates packed BCD (binary coded decimal) numbers, as described in Binary Coded Decimal Variables. If you define binary 10 byte integers, you must write your own routines to use routines in calculations.</p>

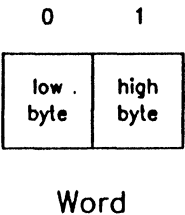


Figure 8–1. Define Word Directive

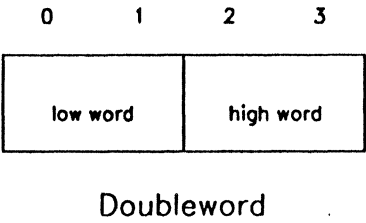


Figure 8-2. Define Doubleword Directive

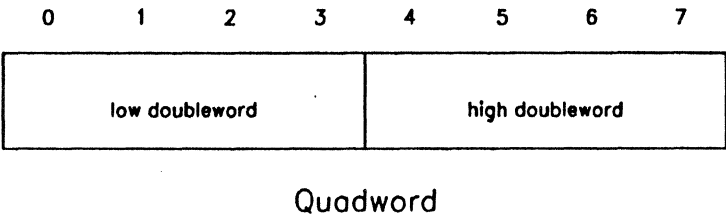


Figure 8-3. Define Quadword Directive

Following are definitions using different directives:

integer	DB	16	; Initialize byte to 16
expression	DW	4*3	; Initialize word to 12
empty	DQ	?	; Allocate uninitialized long
; integer			
	DB	1,2,3,4,5,6	; Initialize six unnamed bytes
high_byte	DD	4294967295	; Initialize double word to
			; 4,294,967,295
tb	DT	2345d	; Initialize 10 byte binary
			; integer

Binary Coded Decimal Variables

Binary coded decimals (BCD) provide a method of doing calculations on large numbers without rounding errors. They are sometimes used in financial applications. There are two kinds: packed and unpacked.

Unpacked BCD numbers are stored one digit to a byte, with the value in the lower four bits. They can be defined with the DB directive. For example, an unpacked BCD number could be defined and initialized as follows:

unpackedr	DB	1,5,8,2,5,2,9	; Initialized to 9,252,851
unpackedf	DB	9,2,5,2,8,5,1	; Initialized to 9,252,851

Whether least significant digits can come either first or last, depends on how you write the calculation routines that handle the numbers. Calculations with unpacked BCD numbers are discussed in Section 18.

Packed BCD numbers are stored two digits to a byte, with one digit in the lower four bits and one in the upper four bits. The leftmost bit holds the sign (0 for positive or 1 for negative). Packed BCD variables can be defined with the DT directive as follows:

packed	DT	9252851	; Allocate 9,252,851
--------	----	---------	----------------------

The 8087 family processors can do fast calculations with packed BCD numbers, as described in Section 21. The 8086 family processors can also do some calculations with packed BCD numbers, but the process is slower and more complicated. See Section 18 for details.

String Variables

Strings are normally initialized with the DB directive. The initializing value is specified as a string constant. Strings can also be initialized by specifying each value in the string. For example, the following definitions are equivalent:

```
version1    DB    97,98,99          ; As ASCII values
version2    DB    'a','b','c'       ; As characters
version3    DB    "abc"             ; As a string
```

One and two character strings (four character strings on the 80386) can also be initialized with any of the other data definition directives. The last (or only) character in the string is placed in the byte with the lowest address. Either 0 or the first character is placed in the next byte. The unused portion of such variables is filled with zeros.

```
asciiz      DB    "<ASM>TEST.ASM",0 ; Use as ASCIIZ string
message     DB    "Enter file name:",0
l_message   EQU    $-message        ;
a_message   EQU    OFFSET message

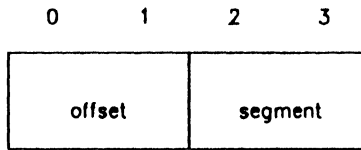
str1        DB    "ab"              ; Stored as 61 62
str2        DD    "ab"              ; Stored as 62 61 00 00
str3        DD    "a"               ; Stored as 61 00 00 00
```

Pointer Variables

Pointer variables (or pointers) are variables that contain the address of a data or code object rather than the object itself. The address in the variable points to another address. Pointers can be either near addresses or far addresses.

Near pointers consist of the offset portion of the address. They can be initialized in word variables by using the DW directive. Values in near address variables can be used in situations where the segment portion of the address is known to be the current segment.

Far pointers consist of both the segment and offset portions of the address. They can be initialized in doubleword variables, using the DD directive. Values in far address variables must be used when the segment portion of the address may be outside the current segment. The segment and offset of a far pointer are stored in the format shown in Figure 8-4.



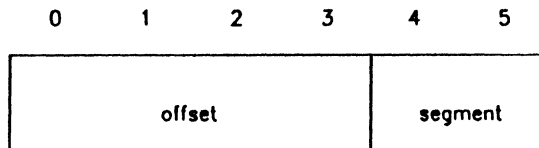
Far Pointer

Figure 8-4. Far Pointer

For example:

```
string      DB      "Text",0           ; Null terminated string
npstring    DW      string              ; Near pointer to string
fpstring    DD      string              ; Far pointer to string
```

Pointers are different on the 80386 processor if the USE32 use type has been specified. In this case the offset portion of an address consists of 32 bits, and the segment portion consists of 16 bits. Therefore a near pointer is 32 bits (a doubleword), and a far pointer is 48 bits (a farword). The segment and offset of a 32 bit mode far pointer are stored in the format shown in Figure 8-5.



Far Pointer in 32-Bit Mode

Figure 8-5. Far Pointer In 32 Bit Mode

For example:

```
_DATA      SEGMENT WORD USE32          PUBLIC 'DATA'
string      DB      "Text",0           ; Null terminated string
npstring     DD      string              ; Near (32 bit) pointer to string
fpstring     .DQ     string              ; Far (48 bit) pointer to string
_DATA      ENDS
```

Real Number Variables

Real numbers must be stored in binary format. However, when initializing variables, you can specify decimal or hexadecimal constants and let the assembler automatically encode them into their binary equivalents. MASM can use two different binary formats for real numbers: IEEE (the default) or Microsoft Binary. You can specify the format by using directives.

This section tells you how to initialize real number variables, describes the two binary formats, and explains real number encoding.

Initializing and Allocating Real Number Variables

Real numbers can be defined by initializing them either with real number constants or with encoded hexadecimal constants. The real number designator (R) must follow numbers specified in encoded format.

The directives for defining real numbers are listed with the sizes of the numbers they can allocate:

Directive	Size
DD	Allocates short (32 bit) real numbers in either the IEEE or Microsoft Binary format.
DQ	Allocates long (64 bit) real numbers in either the IEEE or Microsoft Binary format.
DT	Allocates temporary or 10 byte (80 bit) real numbers. The format of these numbers is similar to the IEEE format. They are always encoded the same regardless of the real number format. Their size is nonstandard and incompatible with Microsoft high level languages. Temporary real format is provided for those who want to initialize real numbers in the format used internally by 8087 family processors.

The 8086 family microprocessors do not have any instructions for handling real numbers. You must write your own routines, use a library that includes real number calculation routines, or use a coprocessor. The 8087 family coprocessors can load real numbers in the IEEE format; they can also use the values in calculations and store the results back to memory, as explained in Section 21.

shrt	DD	98.6	; MASM automatically encodes
long	DQ	5.391E-4	; in current format
ten_byte	DT	-7.31E7	
eshrt	DD	98.6	; encoded in Microsoft
			; Binary format
elong	DQ	3F41AA4C6F445B7Ar	; 5.391E-4 encoded in IEEE
			; format

The real number designator (R) used to specify encoded numbers is explained in Real Number Constants.

Selecting a Real Number Format

MASM can encode four and eight byte real numbers in two different formats: IEEE and Microsoft Binary. Your choice depends on the type of program you are writing. The four primary alternatives are:

- If your program requires a coprocessor for calculations, you must use the IEEE format.
- Most high level languages use the IEEE format. If you are writing modules that will be called from such a language, your program should use the IEEE format. All versions of the C, FORTRAN, and Pascal compilers sold by Microsoft and IBM use the IEEE format.
- If you are writing a module that will be called from most previous versions of Microsoft or IBM BASIC, your program should use the Microsoft Binary format. Versions that support only the Microsoft Binary format include:
 - Microsoft QuickBASIC through Version 2.01
 - Microsoft BASIC Compiler through Version 5.3
 - IBM BASIC Compiler through Version 2.0
 - Microsoft GW-BASIC interpreter (all versions)
 - IBM BASICA interpreter (all versions)

Microsoft QuickBASIC Version 3.0 supports both the Microsoft Binary and IEEE formats as options.

- If you are creating a standalone program that does not use a coprocessor, you can choose either format. The IEEE format is better for overall compatibility with high level languages. The Microsoft Binary format may be necessary for compatibility with existing source code.

Note: *When you interface assembly language modules with high level languages, the real number format only matters if you initialize real number variables in the assembly module. If your assembly module does not use real numbers, or if all real numbers are initialized in the high level language module, the real number format does not make any difference.*

By default, MASM assembles real number data in the IEEE format. This is a change from previous versions of the assembler, which used the Microsoft Binary format by default. If you wish to use the Microsoft Binary format, you must put the `.MSFLOAT` directive at the start of your source file before initializing any real number variables (See Section 6).

Real Number Encoding

The IEEE format for encoding four and eight byte real numbers is illustrated in Figure 8–6.

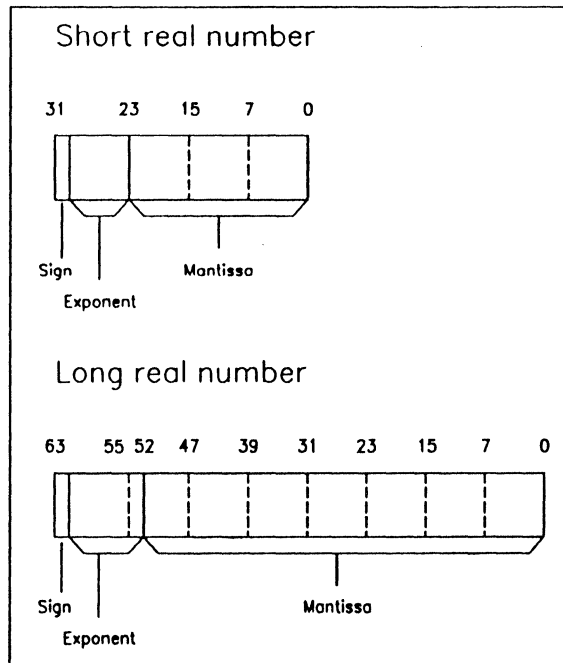


Figure 8–6. Encoding for Real Numbers In IEEE Format

The parts of the real numbers are described as follows:

- Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
- Exponent in the next bits in sequence (8 bits for short real number or 11 bits for long real number).
- All except the first set bit of mantissa in the remaining bits of the variable. Since the first significant bit is known to be set, it need not be actually stored. The length is 23 bits for short real numbers and 52 bits for long real numbers. The Microsoft Binary format for encoding real numbers is illustrated in Figure 8–7.

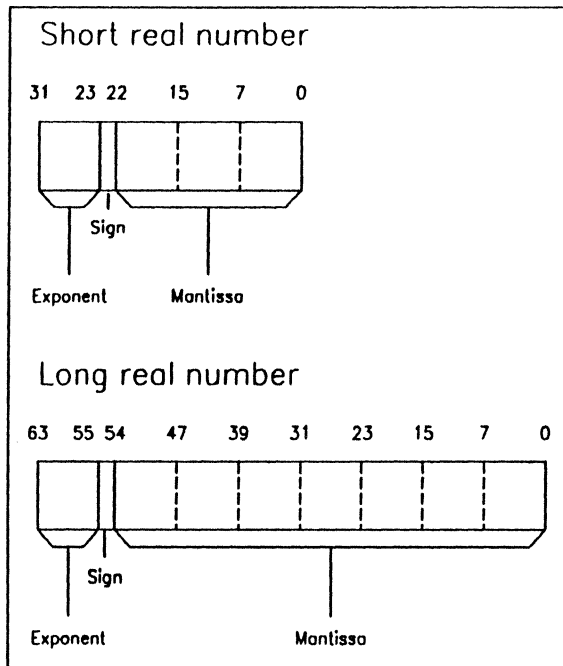


Figure 8–7. Encoding for Real Numbers in Microsoft Binary Format

The three parts of real numbers are described as follows:

- Biased exponent (8 bits) in the high address byte. The bias is 81h for short real numbers and 401h for long real numbers.
- Sign bit (0 for positive or 1 for negative) in the upper bit of the second highest byte.
- All except the first set bit of mantissa in the remaining 7 bits of the second highest byte and in the remaining bytes of the variable. Since the first significant bit is known to be set, it need not be actually stored. The length is 23 bits for short real numbers and 55 bits for long real numbers.

MASM also supports the 10 byte temporary real format used internally by 8087 family coprocessors. This format is similar to IEEE format. The size is nonstandard and is not used by Microsoft compilers or interpreters.

Since the coprocessors can load and automatically convert numbers in the more standard 4 and 8 byte formats, the 10 byte format is seldom used in assembly language programming.

The temporary real format for encoding real numbers is illustrated in Figure 8-8.

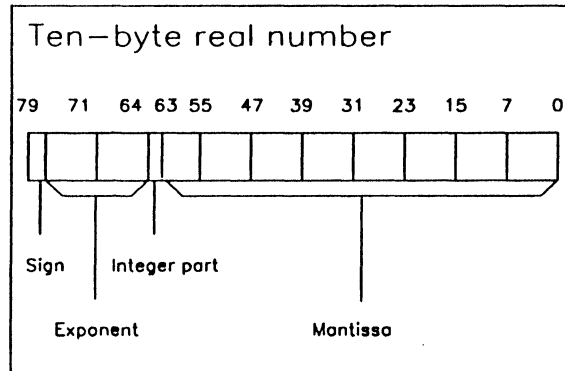


Figure 8-8. Encoding for Real Numbers In Temporary Real Format

The four parts of the real numbers are described as follows:

1. Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
2. Exponent in the next bits in sequence (15 bits for 10 byte real).
3. The integer part of mantissa in the next bit in sequence (bit 63).
4. Remaining bits of mantissa in the remaining bits of the variable. The length is 63 bits.

Notice that the 10 byte temporary real format stores the integer part of the mantissa. This differs from the 4 and 8 byte formats, in which the integer part is implicit.

Arrays and Buffers

Arrays, buffers, and other data structures consisting of multiple data objects of the same size can be defined with the DUP operator. This operator can be used with any of the data definition directives described in this section. The syntax is:

count DUP (*initialvalue*[[*initialvalue*]]...)

The *count* sets the number of times to define *initialvalue*. The initial value can be any expression that evaluates to an integer value, a character constant, or another DUP operator. It can also be the undefined symbol (?) if there is no initial value.

Multiple initial values must be separated by commas. If multiple values are specified within the parentheses, the sequence of values is allocated count times. For example, the statement

```
DB 5 DUP ("Text ")
```

allocates the string "Text" five times for a total of 20 bytes.

DUP operators can be nested up to 17 levels. The initial value (or values) must always be placed within parentheses. For example:

array	DD	10 DUP (1)	; 10 doublewords ; initialized to 1
buffer	DB	256 DUP (?)	; 256 byte buffer
mask1	DB	20 DUP (040h,020h,04h,02h)	; 80 byte buffer ; with bit masks
	DB	32 DUP ("I am here ")	; 320 byte buffer with ; signature for ; debugging
three_d	DD	5 DUP (5 DUP (5 DUP (0)))	; 125 doublewords ; initialized to 0

MASM sometimes generates different object code when the DUP operator is used rather than when multiple values are given. For example, the statement

```
test1      DB      ?,?,?,?,?          ; Indeterminate
```

is indeterminate. It causes MASM to write five zero value bytes to the object file. The statement

```
test2      DB      5 DUP (?)          ; Undefined
```

is undefined. It causes MASM to increase the offset of the next record in the object file by five bytes. Therefore an object file created with the first statement will be larger than one created with the second statement.

In most cases, the distinction between indeterminate and undefined definitions is trivial. The linker adjusts the offsets so that the same executable file is generated in either case. However, the difference is significant in segments with the **COMMON** combine type. If **COMMON** segments in two modules contain definitions for the same variable, one with an indeterminate value and one with an explicit value, the actual value in the executable file varies depending on link order. If the module with the indeterminate value is linked last, the 0 initialized for it overrides the explicit value. You can prevent this by always using undefined rather than indeterminate values in **COMMON** segments. For example, use the first of the following statements:

```
test3      DB      1 DUP (?)          ; Undefined – doesn't initialize
test4      DB      ?                  ; Indeterminate – initializes 0
```

If you use the undefined definition, the explicit value is always used in the executable file regardless of link order.

Labeling Variables

The **LABEL** directive can be used to define a variable of a given size at a specified location. It is useful if you want to refer to the same data as variables of different sizes.

name LABEL type

name is the symbol assigned to the variable, and *type* is the variable size. The type can be any one of the following type specifiers: **BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE**. It can also be the name of a previously defined structure. For example:

```
warray     LABEL    WORD              ; Access array as 50 words
darray     LABEL    DWORD             ; Access same array as 25
                                     ; doublewords
barray     DB       100 DUP(?)        ; Access same array as 100 bytes
```

Setting the Location Counter

The location counter is the value MASM maintains to keep track of the current location in the source file. The location counter is incremented automatically as each source statement is processed. However, the location counter can be set specifically using the ORG directive. The syntax is:

ORG *expression*

Subsequent code and data offsets begin at the new offset specified set by *expression*. The expression must resolve to a constant number. In other words, all symbols used in the expression must be known on the first pass of the assembler.

Note: *The value of the location counter, represented by the dollar sign (\$), can be used in expression, as described in Section 11.*

The following example illustrates one way of assigning symbolic names to absolute addresses. This technique is not possible under protected mode operating systems.

```
; Labeling absolute addresses
STUFF      SEGMENT AT 0           ; Segment has constant value 0
           ORG      410h         ; Offset has constant value 410h
equipment   LABEL    WORD        ; Value at 0000:0410 labeled
                                   ;   equipment
           ORG      417h         ; Offset has constant value 417h
keyboard    LABEL    WORD        ; Value at 0000:0417 labeled
                                   ;   keyboard
STUFF      ENDS

.CODE
.
.
.
ASSUME     ds:STUFF              ; Tell the assembler
mov        ax,STUFF              ; Tell the processor
mov        ds,ax

mov        dx,equipment
mov        keyboard,ax
```

Aligning Data

Some operations are more efficient when the variable used in the operation is lined up on a boundary of a particular size. The `ALIGN` and `EVEN` directives can be used to pad the object file so that the next variable is aligned on a specified boundary. The syntax is:

`EVEN`

`ALIGN number`

The `EVEN` directive always aligns on the next even byte. The `ALIGN` directive aligns on the next byte that is a multiple of *number*. The *number* must be a power of 2. For example, use `ALIGN 2` or `EVEN` to align on word boundaries, or use `ALIGN 4` to align on doubleword boundaries.

If the value of the location counter is not on the specified boundary when an `ALIGN` directive is encountered, the location counter is incremented to a value on the boundary. `NOP` (no operation) instructions are generated to pad the object file. If the location counter is already on the boundary, the directive has no effect.

The `ALIGN` and `EVEN` directives give no efficiency improvements on processors that have an 8 bit data bus (such as the 8088 or 80188). These processors always fetch data one byte at a time, regardless of the alignment. However, using `EVEN` can speed certain operation on processors that have a 16 bit data bus (such as the 8086, 80186, or 80286), since the processor can fetch a word if the data is word aligned, but must do two memory fetches if the data is not word aligned. Similarly, using `ALIGN 4` can speed some operations with a 80386 processor, since the processor can fetch four bytes at a time if the data is doubleword aligned.

The `EVEN` directive should not be used in segments with `BYTE` align type. Similarly, the number specified with the `ALIGN` directive should be at least equal to the size of the align type of the segment where the directive is given.

In the following example, the words at `stuff` and `even stuff` are forced to doubleword boundaries. This makes access to the data faster with processors that have either a 32 bit or 16 bit data bus. Without this alignment, the initial data might start on an odd boundary and the processor would have to fetch half of each word at a time with a 16 bit data bus or half of each doubleword with a 32 bit data bus.

```

.MODEL    small
.STACK   100h
.DATA
.
.
.
stuff    ALIGN    4                ; For faster data access
         DW       66,124,573,99,75
.
.
.
evenstuff ALIGN    4                ; For faster data access
         DW       ?,?,?,?,?
.CODE
start:   mov       ax,@data        ; Load segment location
         mov       ds,ax           ; into DS
         mov       es,ax           ; and ES registers

         mov       cx,5            ; Load count
         mov       si,OFFSET stuff ; Point to source
         mov       di,OFFSET evenstuff ; and destination
         ALIGN     4              ; Align for faster loop
         ; access
mloop:   lodsw      ax             ; Load a word
         inc       ax             ; Make it even by
         ; incrementing
         and       ax,NOT 1       ; and turning off first
         ; bit
         stosw     ; Store
         loop      mloop          ; Again

```

Similarly, the alignment in the code segment speeds up repeated access to the code at the start of the loop. The sample code sacrifices program size in order to achieve significant speed improvements on the 80386 and more moderate improvements on the 8086 and 80286. There is no speed advantage on the 8088.

Section 9

Using Structures and Records

The Macro Assembler can define and use two kinds of multifield variables:

- Structures
- Records

Structures are templates for data objects made up of smaller data objects. A structure can be used to define structure variables, which are made up of smaller variables called fields. Fields within a structure can be different sizes, and each can be accessed individually.

Records are templates for data objects whose bits can be described as groups of bits called fields. A record can be used to define record variables. Each bit field in a record variable can be used separately in constant operands or expressions. The processor cannot access bits individually at run time, but bit fields can be used with logical bit instructions to change bits indirectly.

This section describes structures and records and tells how to use them.

Structures

A structure variable is a collection of data objects that can be accessed symbolically as a single data object. Objects within the structure can have different sizes and can be accessed symbolically. The following subsections describe how to use structure variables:

- Declaring Structure Types
- Defining Structure Variables
- Using Structure Operands

There are two steps in using structure variables:

1. Declare a structure type.

A structure type is a template for data. It declares the sizes and, optionally, the initial values for objects in the structure. By itself, the structure type does not define any data. The structure type is used by MASM during assembly but is not saved as part of the object file.

2. Define one or more variables having the structure type.

For each variable defined, memory is allocated to the object file in the format declared by the structure type.

The structure variable can then be used as an operand in assembler statements. The structure variable can be accessed as a whole by using the structure name, or individual fields can be accessed by using structure and field names.

Declaring Structure Types

The `STRUC` and `ENDS` directives mark the beginning and end of a type declaration for a structure as follows:

```
name STRUC  
fielddeclarations  
name ENDS
```

The *name* declares the name of the structure type. It must be unique. The *fielddeclarations* declare the fields of the structure. Any number of field declarations may be given. They must follow the form of data definitions described in Section 8. Default initial values may be declared individually or with the `DUP` operator.

The names given to fields must be unique within the source file where they are declared. When variables are defined, the field names will represent the offset from the beginning of the structure to the corresponding field.

When declaring strings in a structure type, make sure the initial values are long enough to accommodate the largest possible string. Strings smaller than the field size can be placed in the structure variable, but larger strings will be truncated.

A structure declaration can contain field declarations and comments. Conditional assembly statements are allowed in structure declarations. No other kinds of statements are allowed. Since the STRUC directive is not allowed inside structure declarations, structures cannot be nested.

Note: *The ENDS directive that marks the end of a structure has the same mnemonic as the ENDS directive that marks the end of a segment. The assembler recognizes the meaning of the directive from context. Make sure each SEGMENT directive and each STRUC directive has its own ENDS directive.*

```
student      STRUC                      ; Structure for student records
id           DW      ?                  ; Field for identification #
sname        DB      "Last, First Middle "
scores       DB      10 DUP (100)      ; Field for 10 scores
student      ENDS
```

Within the sample structure student, the fields id, sname, and scores have the offset values 0, 2, and 24, respectively.

Defining Structure Variables

A structure variable is a variable with one or more fields of different sizes. The sizes and initial values of the fields are determined by the structure type with which the variable is defined. The syntax is:

```
[[name]] structurename [[initialvalue [[initialvalue...]]]]
```

name is the name assigned to the variable. If no name is given, the assembler allocates space for the variable, but does not give it a symbolic name. structurename is the name of a structure type previously declared by using the STRUC and ENDS directives.

An initialvalue can be given for each field in the structure. Its type must not be incompatible with the type of the corresponding field. The angle brackets (< >) are required even if no initial value is given. If initial values are given for more than one field, the values must be separated by commas.

If the DUP operator is used to initialize multiple structure variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of structure variables as shown below:

```
war          date          365 DUP (<,,1940>)
```


You need not initialize all fields in a structure. If an initial value is left blank, the assembler automatically uses the default initial value of the field, which was originally determined by the structure type. If there is no default value, the field is undefined.

The following examples use the student type declared in the first example in Declaring Structure Types:

```
s1      student < >           ; Uses default values of type
s2      student < 1467, "White, Robert D.", >
                                ; Override default values of
                                ;   first two fields—use
                                ; default value of third
sarray  student 100 DUP (< >) ; Declare 100 student variables
                                ; with default initial values
```

You cannot initialize any structure field that has multiple values if this field was given a default initial value when the structure was declared. For example, assume the following structure declaration:

```
stuff      STRUC
buffer     DB      100 DUP (?)      ; Can't override
crlf       DB      13,10           ; Can't override
query      DB      'Filename: '    ; String <= can override
endmark    DB      36              ; Can override
stuff      ENDS
```

The buffer and crlf fields cannot be overridden by initial values in the structure definition because they have multiple values. The query field can be overridden as long as the overriding string is no longer than query (10 bytes). A longer string would generate an error. The endmark field can be overridden by any byte value.

Using Structure Operands

Like other variables, structure variables can be accessed by name. Fields within structure variables can also be accessed by using the following:

variable.field

variable must be the name of a structure (or an operand that resolves to the address of a structure). The *field* must be the name of a field within that structure. The *variable* is separated from *field* by a period. The period is discussed as a structure field name operator in Section 11.

The address of a structure operand is the sum of the offsets of *variable* and *field*. The address is relative to the segment or group in which the variable is declared. For example:

```
date          STRUC                      ; Declare structure
month         DB      ?
day           DB      ?
year          DW      ?
date          ENDS

yesterday     .DATA
date          <9,30,1987>                ; Declare structure
today         date    <10,1,1987>        ; variables
tomorrow      date    <10,2,1987>
.CODE
.
.
.
mov     ax,yesterday.day                ; Use structure variables
mov     ah,today.month                 ; as operands
mov     tomorrow.year,dx
mov     bx,OFFSET yesterday            ; Load structure address
mov     ax,[bx].month                  ; Use as indirect operand
.
.
.
```

Records

A record variable is a byte or word variable in which specific bit fields can be accessed symbolically. Records can also be doubleword variables with the 80386 processor. Bit fields within the record can have different sizes. The following subsections describe how to use record variables:

- Declaring Record Types
- Defining Record Variables
- Using Record Operands and Record Variables
- Record Operators
- Using Record Field Operands

There are two steps in declaring record variables:

1. Declare a record type.

A record type is a template for data. It declares the sizes and, optionally, the initial values for bit fields in the record. By itself the record type does not define any data. The record type is used by MASM during assembly but is not saved as part of the object file.

2. Define one or more variables having the record type.

For each variable defined, memory is allocated to the object file in the format declared by the type.

The record variable can then be used as an operand in assembler statements. The record variable can be accessed as a whole by using the record name, or individual fields can be specified by using the record name and a field name combined with the field name operator. A record type can also be used as a constant (immediate data).

Declaring Record Types

The **RECORD** directive declares a record type for an 8 or 16 bit record that contains one or more bit fields. With the 80386, 32 bit records can also be declared. The syntax is:

recordname **RECORD** *field* [*,field...*]

The *recordname* is the name of the record type to be used when creating the record. The *field* declares the name, width, and initial value for the field.

The syntax for each *field* follows:

fieldname:width [*=expression*]

fieldname is the name of a field in the record, *width* is the number of bits in the field, and *expression* is the initial (or default) value for the field.

Any number of field combinations can be given for a record, as long as each is separated from its predecessor by a comma. The sum of the widths for all fields must not exceed 16 bits.

width must be a constant. If the total width of all declared fields is larger than eight bits, then the assembler uses two bytes; Otherwise, only one byte is used.

Note: *Records can be up to 32 bits in width when the 80386 processor is enabled with .386. If the total width is 8 bits or less, the assembler uses 1 byte; if the width is 9 to 16 bytes, the assembler uses 2 bytes; and if the width is larger than 16 bits, the assembler uses 4 bytes.*

If *expression* is given, it declares the initial value for the field. An error message is generated if an initial value is too large for the width of its field. If the field is at least seven bits wide, you can use an ASCII character for *expression*. The *expression* must not contain a forward reference to any symbol.

In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total 8 bits or 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits in the high end of the record are initialized to 0.

The following example creates a byte record type `color` having four fields: `blink`, `back`, `intense`, and `fore`. The contents of the record type are shown in Figure 9-1.

```
color      RECORD      blink:1,back:3,intense:1,fore:3
```

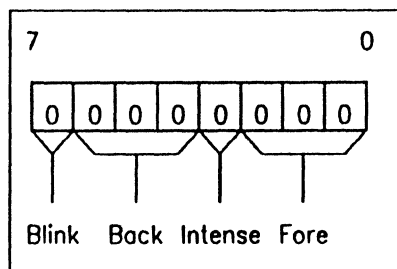


Figure 9-1. Byte record

Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. No data are created.

The following example creates a record type `cw` having six fields. Each record declared by using this type occupies 16 bits of memory. Figure 9–2 shows the contents of the record type.

```
cw      RECORD      r1:3=0,ic:1=0,rc:2=0,pc:2=3,r2:2=1,masks:6=63
```

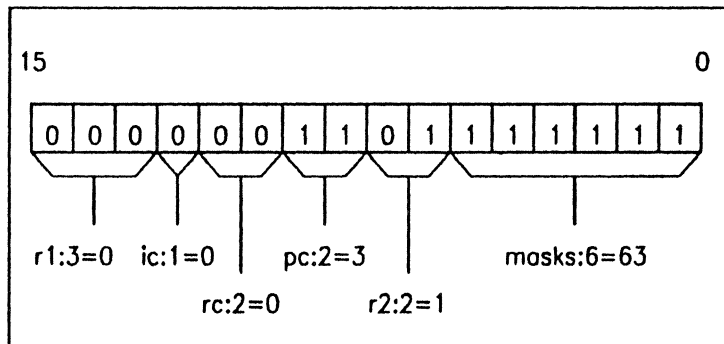


Figure 9–2. 16 Bit Record

Default values are given for each field. They can be used when data is declared for the record.

Defining Record Variables

A record variable is an 8 bit or 16 bit variable whose bits are divided into one or more fields. With the 80386, 32 bit variables are also allowed. The syntax is:

```
[[name]] recordname [[initialvalue [[initialvalue...]]]]
```

The *name* is the symbolic name of the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *recordname* is the name of a record type that was previously declared by using the `RECORD` directive.

An *initialvalue* for each field in the record can be given as an integer, character constant, or an expression that resolves to a value compatible with the size of the field. Angle brackets (< >) are required even if no initial value is given. If initial values for more than one field are given, the values must be separated by commas.

If the DUP operator (see Section 8) is used to initialize multiple record variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of record variables as shown below:

```
xmas          color          50 DUP (<1,2,0,4>)
```

You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is declared by the record type. If there is no default value, each bit in the field is cleared.

Using Record Operands and Record Variables and Record Operators illustrate ways to use record data after it has been declared.

The following definition creates a variable named warning whose type is given by the record type color:

```
color      RECORD    blink:1,back:3,intense:1,fore:3    ; Record
                                                    ; declaration
warning    color      <1,0,1,4>                          ; Record
                                                    ; definition
```

The initial values of the fields in the variable are set to the values given in the record definition. The initial values would override the default record values, had any been given in the declaration. The contents of the record variable are shown in Figure 9-3.

The following example creates an array named colors containing 16 variables of type color

```
color      RECORD    blink:1,back:3,intense:1,fore:3    ; Record
                                                    ; declaration
colors     color      16 DUP (<>)                       ; Record
                                                    ; declaration
```

Since no initial values are given in either the declaration or the definition, the variables have undefined (0) values.

The following example creates a variable named newcw with type cw:

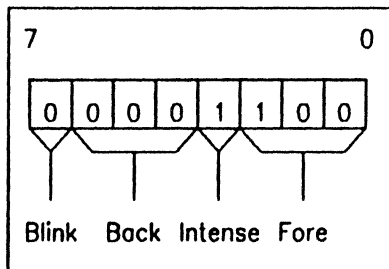


Figure 9-3. Byte Record

```

cw      RECORD      r1:3=0,ic:1=0,rc:2=0,pc:2=3,r2:2=1,masks:6=63
newcw   cw          <,,2,,>

```

The default values set in the type declaration are used for all fields except the rc field. This field is set to 2. The contents of the variable are shown in Figure 9-4.

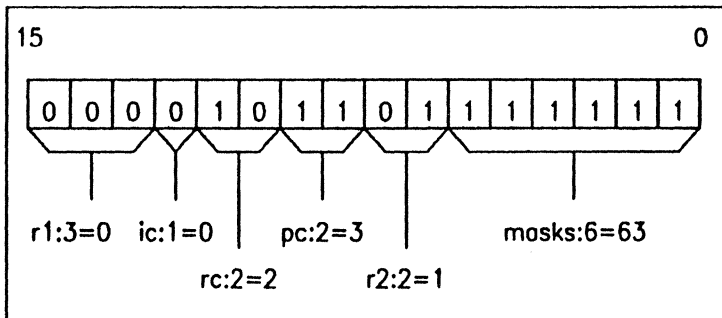


Figure 9-4. Type Declaration

Using Record Operands and Record Variables

A record operand refers to the value of a record type. Do not confuse it with a record variable. A record operand is a constant; a record variable is a value stored in memory. A record operand can be used with the following syntax:

recordname [[[*value*][,*value*...]]]

recordname must be the name of a record type declared in the source file. The optional *value* is the value of a field in the record. If more than one *value* is given, each value must be separated by a comma. Values can include expressions or symbols that evaluate to constants. The enclosing angle brackets (< >) are required, even if no value is given. If no value for a field is given, the default value for that field is used.

In the following example, the record operand `color <0,3,0,2>` and the record variable `warning` are loaded into registers. For example:

```
.DATA
color      RECORD    blink:1,back:3,intense:1,fore:3 ; Record
                                                    ; declaration
window     color     <0,6,1,6>                        ; Record
                                                    ; definition

.CODE
.
.
.
mov        ah,color <0,3,0,2>                        ; Load record operand
                                                    ; (constant value 32h)
mov        bh,window                                ; Load record variable
                                                    ; (memory value 6Eh)
```

The contents of the values are shown in Figure 9–5.

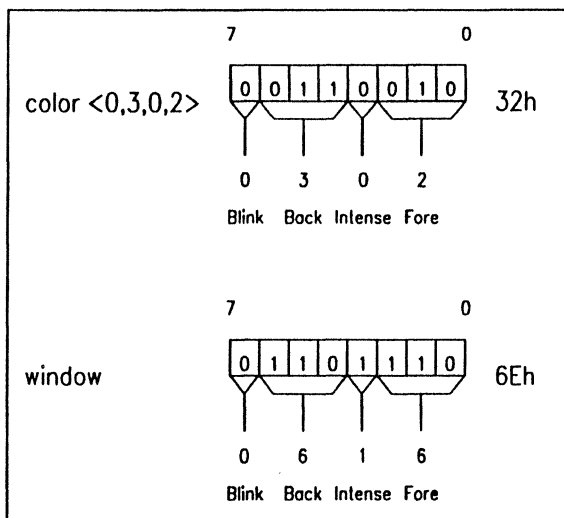


Figure 9-5. Record Operand Values

Record Operators

The WIDTH and MASK operators are used exclusively with records to return constant values representing different aspects of previously declared records.

The MASK Operator

The MASK operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a field bit. All other bits contain 0.

`MASK {recordfieldname | record}`

recordfieldname may be the name of any field in a previously defined record. *record* may be the name of any previously defined record. The NOT operator is sometimes used with the MASK operator to reverse the bits of a mask. For example:

color	.DATA	blink:1,back:3,intense:1,fore:3	
message	RECORD	color	<0,5,1,1>
	.CODE		
	.		
	.		
	mov	ah,message	; Load initial 0101 1001
	and	ah,NOT MASK back	; Turn off AND 1000 1111
			; back
			; 0000 1001
	or	ah,MASK blink	; Turn on OR 1000 0000
			; blink
			; 1000 1001
	xor	ah,MASK intense	; Toggle X OR 0000 1000
			; intense
			; 1000 0001

The WIDTH Operator

The WIDTH operator returns the width (in bits) of a record or record field and uses the following syntax:

WIDTH {*recordfieldname* | *record*}

recordfieldname may be the name of any field defined in any record.

record may be the name of any defined record. Note that the width of a field is the number of bits assigned for that field; the value of the field is the starting position (from the right) of the field. For example:

```
.DATA
color      RECORD    blink:1,back:3,intense:1,fore:3

wblink     EQU       WIDTH blink    ; wblink    = 1  blink    = 7
wback      EQU       WIDTH back     ; wback     = 3  back     = 4
wintense   EQU       WIDTH intense  ; wintense  = 1  intense  = 3
wfore      EQU       WIDTH fore     ; wfore     = 3  fore     = 0
wcolor     EQU       WIDTH color    ; wcolor    = 8

prompt     color      <1,5,1,1>

.CODE
.
.
.
IF         (WIDTH color) GE 8 ; If color is 16 bit, load
mov        ax,prompt         ; into 16 bit register
ELSE
mov        al,prompt         ; load into low 8 bit
xor        ah,ah             ; register
                                ; and clear high 8 bit
                                ; register
ENDIF
```

Using Record Field Operands

Record field operands represent the location of a field in its corresponding record. The operand evaluates to the bit position of the low order bit in the field and can be used as a constant operand. The field name must be from a previously declared record.

Record field operands are often used with the **WIDTH** and **MASK** operators, as described in *The Mask Operator* and *The Width Operator*.

The following example illustrates several ways in which record fields can be used as operands and in expressions.

```

        .DATA
color      RECORD blink:1,back:3,intense:1,fore:3    ; Record declaration
cursor color <1,5,1,1>                               ; Record definition
        .CODE
        .
        .
        .
; Rotate back of cursor without changing other values

        mov     al,cursor          ; Load value from memory
        mov     ah,al              ; Save a copy for work      1101 1001=ah/al
        and     al,NOT MASK back   ; Mask out old bits      and 1000 1111=mask
                                   ; to save old cursor
                                   ;
                                   ;      1000 1001=a1
        mov     cl,back            ; Load bit position
        shr     ah,cl              ; Shift to right          0000 1101=ah
        inc     ah                 ; Increment                0000 1110=ah

        shl     ah,cl              ; Shift left again         1110 0000=ah
        and     ah,MASK back       ; Mask off extra bits and 0111 0000=mask
                                   ; to get new cursor
                                   ;
                                   ;      0110 0000 ah
        or      ah,al              ; Combine old and new or 1000 1001 a1
                                   ;
        mov     cursor,ah          ; Write back to memory    1110 1001 ah

```


Section 10

Creating Programs from Multiple Modules

Most medium and large assembly-language programs are created from several source files or modules. When several modules are used, the scope of symbols becomes important. This section discusses the scope of symbols and explains how to declare global symbols that can be accessed from any module. It also tells you how to specify a module that will be accessed from a library. It includes the following subsections:

- Declaring Symbols Public
- Declaring Symbols External
- Using Multiple Modules
- Declaring Symbols Communal
- Specifying Library Files

Symbols such as labels and variable names can be either local or global in scope. By default, all symbols are local; they are specific to the source file in which they are defined. Symbols must be declared global if they must be accessed from modules other than the one in which they are defined.

To declare symbols global, they must be declared public in the source module in which they are defined. They must also be declared external in any module that must access the symbol. If the symbol represents uninitialized data, it can be declared communal—meaning that the symbol is both public and external. The PUBLIC, EXTRN, and COMM directives are used to declare symbols public, external, and communal, respectively.

Note: *The term local has a different meaning in assembly language than in many high-level languages. Often, local symbols in compiled languages are symbols that are known only within a procedure (called a function, routine, subprogram, or subroutine, depending on the language). Local symbols of this type cannot be declared by MASM, although procedures can be written to allocate local symbols dynamically at run time, as described in Section 19.*

By default, the assembler converts all lowercase letters in names declared with the PUBLIC, EXTRN, and COMM directives to uppercase letters before copying the name to the object file. The /ML and /MX options can be used in the MASM command line to direct the assembler to preserve lowercase letters when copying public and external symbols to the object file. This should be done when preparing assembler modules to be linked with modules from case-sensitive languages such as C.

Declaring Symbols Public

The PUBLIC directive is used to declare symbols public so that they can be accessed from other modules. If a symbol is not declared public, the symbol name is not written to the object file. The symbol has the value of its offset address during assembly, but the name and address are not available to the linker.

If the symbol is declared public, its name is associated with its offset address in the object file. During linking, symbols in different modules—but with the same name—are resolved to a single address.

Public symbol names are also used by some symbolic debuggers. The syntax is:

```
PUBLIC name[ ,name ] ...
```

The name must be the name of a variable, label, or numeric equate defined within the current source file. PUBLIC declarations can be placed anywhere in the source file. Equate names, if given, can only represent 1- or 2-byte integer or string values. Text macros (or text equates) cannot be declared public.

Although absolute symbols can be declared public, aliases for public symbols can cause errors. For example, the following statements are illegal:

```
PUBLIC  lines      ; Declare absolute symbol public
lines EQU rows    ; Declare alias for lines
rows EQU 25       ; Illegal - Assign value to alias
```

Following is an example of symbols declared PUBLIC:

```

PUBLIC      true,status,first,clear
.MODEL     small
true      EQU      -1
          .DATA
status    DB        1
          .CODE
          .
          .
          .
first     LABEL     FAR
clear     PROC
          .
          .
          .
clear     ENDP
```


Declaring Symbols External

If a symbol undeclared in a module must be accessed by instructions in that module, it must be declared with the EXTRN directive.

This directive tells the assembler not to generate an error, even though the symbol is not in the current module. The assembler assumes that the symbol occurs in another module. However, the symbol must actually exist and must be declared public in some module. Otherwise, the linker generates an error. The syntax is:

```
EXTRN name:type [ , name:type ] ...
```

The EXTRN directive defines an external variable, label, or symbol of the specified *name* and *type*. The *type* must match the type given to the item in its actual definition in some other module. It can be any one of the following:

Description	Types
Distance specifier	NEAR, FAR, or PROC
Size specifier	BYTE, WORD, DWORD, FWORD, QWORD, or TBYTE
Absolute	ABS

The ABS type is for symbols that represent constant numbers, such as equates declared with the EQU and = directives (see Section 13).

The PROC type represents the default type for a procedure. For programs that use simplified segment directives, the type of an external symbol declared with PROC will be near for small or compact model, or far for medium, large, or huge model. Section 7 describes how to declare the memory model using the .MODEL directive. If full segment definitions are used, the default type represented by PROC is always near.

Although the actual address of an external symbol is not determined until link time, the assembler assumes a default segment for the item, based on where the EXTRN directive is placed in the source code. Placement of EXTRN directives should follow these rules.

- NEAR code labels (such as procedures) must be declared in the code segment from which they are accessed.
- FAR code labels can be declared anywhere in the source code. It may be convenient to declare them in the code segment from which they are accessed if the label may be FAR in one context or NEAR in another.
- Data must be declared in the segment in which it occurs. This may require that you define a dummy data segment for the external declaration.
- Absolute symbols can be declared anywhere in the source code.

The following example shows how each type of external symbol could be declared and used in a small-model program that uses simplified segment directives.

```

                                EXTRN      max:ABS,act:FAR          ; Constant or FAR label
                                                                    ;      anywhere
                                DOSSEG
                                .MODEL      small
                                .STACK 100h
                                .DATA
                                EXTRN      nvar:BYTE                ; NEAR variable in near data
                                .FARDATA
                                EXTRN      fvar:WORD                ; FAR variable in far data
                                .CODE
                                EXTRN      task:PROC                 ; PROC or NEAR in near code
start:  mov      ax,@data      ; Load segment
        mov      ds,ax        ; into DS
        ASSUME    es:SEG fvar ; Tell assembler
        mov      ax,SEG fvar  ; Tell processor that ES
        mov      es,ax        ; has far data segment
        .
        .
        mov      ah,nvar      ; Load external NEAR
                                ;      variable
        mov      bx,fva       ; Load external FAR variable
        mov      cx,max       ; Load external constant
        call     task         ; Call procedure (NEAR or
                                ;      FAR)
        jmp      act          ; Jump to FAR label
                                ;
                                END      start

```

Notice the use of the PROC type specifier to make the external-procedure memory model independent. The jump and its external declaration are written so that they will be FAR regardless of the memory model. Using these techniques, you can change the memory model without breaking code.

The following example shows a fragment similar to the one in the previous example, but with full segment definitions. Notice that the types of code labels must be declared specifically. If you wanted to change the memory model, you would have to specifically change each external declaration and each call or jump.

```

STACK      EXTRN      max:ABS,act:FAR          ; Constant or FAR label anywhere
            SEGMENT    PARA STACK 'STACK'
            DB          100h DUP (?)
STACK
-Data      ENDS
-Data      SEGMENT    WORD PUBLIC 'DATA'
            EXTRN      nvar:BYTE              ; NEAR variable in near
                                                ; data
-Data ENDS
FAR-Data   SEGMENT    PARA 'FAR-Data'
            EXTRN      fvar:WORD              ; FAR variable in far
                                                ; data
FAR DATA ENDS
DGROUP     GROUP      —Data,STACK
-TEXT      SEGMENT    BYTE PUBLIC 'CODE'
            EXTRN      task:NEAR              ; NEAR procedure in near
                                                ; code
start:      ASSUME     cs:-TEXT,ds:DGROUP,ss:DGROUP
            mov        ax,DGROUP              ; Load segment
            mov ds,    ax                      ; into DS
            ASSUME     es:SEG fvar             ; Tell assembler
            mov        ax,SEG fvar             ; Tell processor that ES
            mov        es,ax                   ; has far data segment
            .
            .
            mov        ah,nvar                 ; Load external NEAR
                                                ; variable
            mov        bx,fvar                 ; Load external FAR
                                                ; variable
            mov        cx,max                  ; Load external constant
            call task                           ; Call NEAR procedure
                                                ; Jump to FAR label
-TEXT      jmp act
            ENDS
            END      start

```

Using Multiple Modules

The following source files illustrate a program that uses public and external declarations to access instruction labels. The program consists of two modules called hello and display

The hello module is the program's initializing module. Execution starts at the instruction labeled start in the hello module. After initializing the data segment, the program calls the procedure display in the display module, where a CTOS call is used to display a message on the screen. Execution then returns to the address after the call in the hello module.

The hello module is shown here:

PAGE 64,132

```

                                TITLE      HELLO1
                                .MODEL      small
                                .STACK      256
                                .DATA
message      PUBLIC      message, lmessage
lmessage     DB          "Hello, world.",13,10
                                EQU         $ - message
                                .CODE
Start:
                                EXTRN      display: PROC          ; Declare in near code segment
                                call display                        ; Call other module
                                EXTRN      ErrorExit: FAR          ; OS far call
                                push       ax
                                call       ErrorExit              ; Call CTOS to exit
                                END        Start                  ; Start address in main module

```

The display module is shown here:

```

        EXTRN        WriteBsRecord: FAR

        TITLE DISPLAY
        .MODEL small

        EXTRN        lmessage: ABS           ; Declare anywhere
        .DATA
        EXTRN        message: BYTE          ; Declare in near data
segment

; We write to video using SAM's pre-opened bytestream
; which is located in the data segment.

EXTRN        bsVid: BYTE

cbWrittenRet    DW        ?
                .286                               ; allow protected mode
opcodes

        .CODE
PUBLIC display
display PROC
; Here we will print the message using CTOS call WriteBsRecord
        push    ds                                ; 1st arg is pbsVid
        push    OFFSET bsVid
        push    ds                                ; 2nd arg is prgcsMsg
        push    OFFSET message
        push    lmessage                          ; 3rd arg is cbMsg
        push    ds                                ; 4th arg is pcbWrittenRet
        push    OFFSET cbWrittenRet
        call    WriteBsRecord                    ; make the call
        ret
display ENDP
END
                ; No start address in
                ; second module
```

The sample program is a variation of the hello.asm program used in examples in Section 3, except that it uses an external procedure to display to the screen. Notice that all symbols defined in one module but used in another are declared PUBLIC in the defining module and declared EXTRN in the using module.

For instance, message and lmessage are declared PUBLIC in hello and declared EXTRN in display. The procedure display is declared EXTRN in hello and PUBLIC in display

To create an executable file for these modules, assemble each module separately, as in the following command lines:

```
MASM
[args] hello;

MASM
[args] display;
```

Then link the two modules:

```
Link
Object Modules      hello.obj display.obj
Run file            hello.run
[Map file]
[Publics?]
[Line numbers]
[Stack size]
[Max array, data, code]
[Min array, data, code]
[Runfile Mode]
[Version]
[Libraries]
[DS allocation?]
[Symbol file]
```

The result is the executable file hello.run.

For each source module, MASM writes a module name to the object file. The module name is used by some debuggers and by the linker when it displays error messages. The module name is always the base name of the source module file.

For compatibility, MASM recognizes the NAME directive. However, NAME has no effect. Arguments to the directive are ignored.

Declaring Symbols Communal

Communal variables are uninitialized variables that are both public and external. They are often declared in include files.

If a variable must be used by several assembly routines, you can declare the variable communal in an include file, and then include the file in each of the assembly routines. Although the variable is declared in each source module, it exists at only one address. Using a communal variable in an include file and including it in several source modules is an alternative to defining the variable and declaring it public in one source module and then declaring it external in other modules.

If a variable is declared communal in one module and public in another, the public declaration takes precedence and the communal declaration has the same effect as an external declaration. The syntax is:

COMM *definition* [*,definition*] ...

Each *definition* has the following syntax:

[[NEAR | FAR] *label*:*size* [:*count*]

A communal variable can be NEAR or FAR. If neither is specified, the type will be that of the default memory model. If you use simplified segment directives, the default type is NEAR for small and medium models, or FAR for compact, large, and huge models. If you use full segment definitions the default type is NEAR.

label is the name of the variable. *size* can be BYTE, WORD, DWORD, QWORD, or TBYTE. *count* is the number of elements. If no count is given, one element is assumed. Multiple variables can be defined with one COMM statement by separating each variable with a comma.

Note: *C variables declared outside functions (except static variables) are communal unless explicitly initialized; they are the same as assembly-language communal variables. If you are writing assembly-language modules for C, you can declare the same communal variables in C include files and in MASM include files.*

MASM cannot tell whether a communal variable has been used in another module. Allocation of communal variables is handled by LINK. As a result, communal variables have the following limitations that other variables declared in assembly language do not have:

- Communal variables cannot be initialized.

Under DOS, initial values are not guaranteed to be 0 or any other value. The variables can be used for data, such as file buffers, that are not given a value until run time.

- Communal variables are not guaranteed to be allocated in the sequence in which they are declared.

Assembly-language techniques that depend on the sequence and position in which data is defined should not be used with communal variables. For example, the following statements do not work:

```

1buffer    COMM    buffer:WORD    :128
            EQU     $ - buffer    ; 1buffer won't have desired
                                   ; value
bbuffer    LABEL   BYTE           ; bbuffer won't have desired
                                   ; address
            COMM    wbuffer:WORD:128
    
```

- Placement of communal declarations follows the same rules as external declarations.

They must be declared inside a data segment. Examples of near and far communal variables are shown below:

```

.DATA
COMM NEAR    nbuffer:BYTE:30
.FARDATA
COMM FAR     fbuffer:WORD:40
    
```

- Communal variables are allocated in segments that are part of the Microsoft segment conventions. You cannot override the default to place communal variables in other segments.

Near communal variables are placed in a segment called `c common`, which is part of `DGROUP`. This group is created and initialized automatically if you use simplified segment directives. If you use full segment directives, you must create a group called `DGROUP` and use the `ASSUME` directive to associate it with the `DS` register.

Far communal variables are placed in a segment called `FAR BSS`. This segment has combine type `private` and class type `'FAR BSS'`. This means that multiple segments with the same name can be created. Such segments cannot be accessed by name. They must be initialized indirectly using the `SEG` operator. For example, if a far communal variable (with word size) is called `comvar`, its segment can be initialized with the following lines:

```

ASSUME ds:SEG comvar    ; Tell the assembler
mov     ax,SEG comvar    ; Tell the processor
mov     ds,ax
mov     bx,comvar        ; Use the variable
    
```


The following example creates two communal variables. The first is a word variable called var. The second is a 10-byte array called buffer. Both have the default size associated with the memory model of the program in which they are used.

```
IF      @datasize
.FARDATA
ELSE
.DATA
ENDIF
COMM   var:WORD, buffer:BYTE:10
```

Section 11

Using Operands and Expressions

The following subsections explain the use of operands and expressions:

- Using Operands with Directives
- Using Operators
- Using the Location Counter
- Using Forward References
- Strong Typing for Memory Operands

Operands are the arguments that define values to be acted on by instructions or directives. Operands can be constants, variables, expressions, or keywords, depending on the instruction or directive, and the context of the statement.

A common type of operand is an expression. An expression consists of several operands that are combined to describe a value or memory location. Operators indicate the operations to be performed when combining the operands of an expression.

Expressions are evaluated at assembly time. By using expressions, you can instruct the assembler to calculate values that would be difficult or inconvenient to calculate when you are writing source code.

This section covers operands, expressions, and operators as they are evaluated at assembly time. See Section 16 for a discussion of the addressing modes that can be used to calculate operand values at run time. This section also discusses the location counter operand, forward references, and strong typing of operands.

Using Operands with Directives

Each directive requires a specific type of operand. Most directives take string or numeric constants, or symbols or expressions that evaluate to such constants.

The type of operand varies for each directive, but the operand must always evaluate to a value that is known at assembly time. This differs from instructions, whose operands may not be known at assembly time and may vary at run time. Operands used with instructions are covered in Section 16.

Some directives, such as those used in data declarations, accept labels or variables as operands. When a symbol that refers to a memory location is used as an operand to a directive, the symbol represents the address of the symbol rather than its contents. This is because the contents may change at run time and are therefore not known at assembly time.

In the following example, the operand of the DW directive in the third statement represents the address of var (100h) rather than its contents (10h). The address is relative to the start of the segment in which var is defined.

	ORG	100h	; Set address to 100h
var	DB	10h	; Address of var is 100h
			; Value of var is 10h
pvar	DW	var	; Address of pvar is 101h
			; Value of pvar is
			; address of var (100h)

The following example illustrates the different kinds of values that can be used as directive operands:

_TEXT	TITLE	doit	; String
	SEGMENT	BYTE PUBLIC 'CODE'	; Key words
	INCLUDE	<include>bios.inc	; Pathname
	.RADIX	16	; Numeric constant
t	DW	a / b	; Numeric expression
	PAGE	+	; Special character
sum	EQU	x * y	; Numeric expression
here	LABEL	WORD	; Type specifier

Using Operators

This subsection describes the use of the following operators:

- Calculation Operators
- Relational Operators
- Segment Override Operators
- Type Operators
- Operator Precedence

The assembler provides a variety of operators for combining, comparing, changing, or analyzing operands. Some operators work with integer constants, some with memory values, and some with both. Operators cannot be used with floating point constants since MASM does not recognize real numbers in expressions.

It is important to understand the difference between operators and instructions. Operators handle calculations of constant values that are known at assembly time. Instructions handle calculations of values that may not be known until run time. For example, the addition operator (+) handles assembly time addition, while the ADD and ADC instructions handle run time addition.

This subsection describes the different kinds of operators used in assembly language statements and gives examples of expressions formed with them. In addition to the operators described in this subsection, you can use the DUP operator, the record operators, and the macro operators.

Calculation Operators

MASM provides the common arithmetic operators as well as several other operators for adding, shifting, or doing bit manipulations. The subsections below describe operators that can be used for doing numeric calculations. These are:

- Arithmetic
- Structure Field Name
- Index
- Shift
- Bitwise Logical

Note: *MASM does 32 bit arithmetic on expressions when the 80386 is enabled and 16 bit arithmetic when it is not. Constant values used with calculation operators are extended to 17 bits (33 bits with 80386 enabled) before the calculations are done.*

Arithmetic Operators

MASM recognizes a variety of arithmetic operators for common mathematical operations. Table 11–1 lists the arithmetic operators.

Table 11–1. Arithmetic Operators

Operator	Syntax	Meaning
+	<i>+ expression</i>	Positive (unary)
–	<i>–expression</i>	Negative (unary)
*	<i>expression1*expression2</i>	Multiplication
/	<i>expression1/expression2</i>	Integer division
MOD	<i>expression1MODexpression2</i>	Remainder (modulus)
+	<i>expression1 + expression2</i>	Addition
–	<i>expression1–expression2</i>	Subtraction

For all arithmetic operators except the addition operator (+) and the subtraction operator(-), the expressions operated on must be integer constants.

The addition and subtraction operators can be used to add or subtract an integer constant and a memory operand. The result can be used as a memory operand.

The subtraction operator can also be used to subtract one memory operand from another, but only if the operands refer to locations within the same segment. The result will be a constant, not a memory operand.

Note: *The unary plus and minus (used to designate positive or negative numbers) are not the same as the binary plus and minus (used to designate addition or subtraction). The unary plus and minus have a higher level of precedence, as described in Operator Precedence.*

The following example illustrates arithmetic operators used in integer expressions.

intgr	=	14 * 3	; = 42
intgr	=	intgr / 4	; 42 / 4 = 10
intgr	=	intgr MOD 4	; 10 mod 4 = 2
intgr	=	intgr + 4	; 2 + 4 = 6
intgr	=	intgr - 3	; 6 - 3 = 3
intgr	=	-intgr - 8	; -3 - 8 = -11
intgr	=	-intgr - intgr	; 11 - -11 = 22

The following example illustrates arithmetic operators used in memory expressions.

	ORG	100h	
a	DB	?	; Address is 100h
b	DB	?	; Address is 101h
mem1	EQU	a + 5	; mem1 = 100h + 5 = 105h
mem2	EQU	a - 5	; mem2 = 100h - 5 = 0FBh
const	EQU	b - a	; const = 101h - 100h = 1

Structure Field Name Operator

The structure field name operator (.) indicates addition. It is used to designate a field within a structure. The syntax is:

variable.field

The variable is a memory operand (usually a previously declared structure variable) and field is the name of a field within the structure. See Section 9 for more information.

```
.DATA
date      STRUC                               ; Declare structure
month     DB      ?
day       DB      ?
year      DW      ?
date      ENDS
yesterday date    <12,31,1987>                ; Define structure
                                                ; variables
today date <1,1,1988>

.CODE
.
.
.
mov      bh,yesterday.day                    ; Load structure variable
mov      bx,OFFSET today                     ; Load structure variable
                                                ; address
inc      [bx].year                           ; Use in indirect memory
                                                ; operand
```

Index Operator

The index operator ([]) indicates addition. It is similar to the addition (+) operator. The syntax is:

[[expression1][expression2]

In most cases, *expression1* is simply added to *expression2*. The limitations of the addition operator for adding memory operands also apply to the index operator. For example, two direct memory operands cannot be added. The expression `label1[label2]` is illegal if both are memory operands.

The index operator has an extended function in specifying indirect memory operands. Section 16 explains the use of indirect memory operands. The index brackets must be outside the register or registers that specify the indirect displacement. However, any of the three operators that indicate addition (the addition operator, the index operator, or the structure field name operator) may be used for multiple additions within the expression.

For example, the following statements are equivalent:

```
mov    ax,table[bx][di]
mov    ax,table[bx+di]
mov    ax,[table+bx+di]
mov    ax,[table][bx][di]
```

The following statements are illegal because the index operator does not enclose the registers that specify indirect displacement:

```
mov    ax,table+bx+di           ; Illegal – no index
                                   ; operator
mov    ax,[table]+bx+di         ; Illegal – registers not
                                   ; inside index operator
```

The index operator is typically used to index elements of a data object, such as variables in an array or characters in a string.

The following example illustrates the index operator used with direct memory operands.

```
mov    al,string[3]             ; Get 4th element of
                                   ; string
add    ax,array[4]              ; Add 5th element of array
mov    string[7],al             ; Load into 8th element of
                                   ; string
```

The following example illustrates the index operator used with indirect memory operands.

```
mov    ax,[bx]                  ; Get element BX points to
add    ax,array[si]             ; Add element SI points to
mov    string[di],al            ; Load element DI points
                                   ; to
cmp    cx,table[bx][di]         ; Compare to element BX
                                   ; and DI
                                   ; point to
```


Shift Operators

The SHR and SHL operators can be used to shift bits in constant values. Both perform logical shifts. Bits on the right for SHL and on the left for SHR are zero filled as their contents are shifted out of position. The syntax is:

expression SHR *count*

expression SHL *count*

expression is shifted right or left by *count* number of bits. Bits shifted off either end of the expression are lost. If *count* is greater than or equal to 16 (32 on the 80386), the result is 0.

Do not confuse the SHR and SHL operators with the processor instructions having the same names. The operators work on integer constants only at assembly time. The processor instructions work on register or memory values at run time. The assembler can tell the difference between instructions and operands from context.

```
mov     ax,01110111b SHL 3 ; Load 01110111000b
mov     ah,01110111b SHR 3 ; Load 01110b
```

Bitwise Logical Operators

The bitwise operators perform logical operations on each bit of an expression. The expressions must resolve to constant values.

Table 11–2 lists the logical operators and their meanings.

Table 11–2. Logical Operators.

Operator	Syntax	Meaning
NOT	NOT <i>expression</i>	Bitwise complement
AND	<i>expression1</i> AND <i>expression2</i>	Bitwise AND
OR	<i>expression1</i> OR <i>expression2</i>	Bitwise inclusive OR
XOR	<i>expression1</i> XOR <i>expression2</i>	Bitwise exclusive OR

Do not confuse the NOT, AND, OR, and XOR operators with the processor instructions having the same names. The operators work on integer constants only at assembly time. The processor instructions work on register or memory values at run time. The assembler can tell the difference between instructions and operands from context.

Note: *Although calculations on expressions using the AND, OR, and XOR operators are done using 17 bit numbers (33 bit with .386), the results are truncated to 16 bits (32 bits with .386).*

Following are examples of bitwise logical operators:

mov	ax,NOT 11110000b	; Load
		; 1111111100001111b
mov	ah,NOT 11110000b	; Load 00001111b
mov	ah,01010101b AND 11110000b	; Load 01010000b
mov	ah,01010101b OR 11110000b	; Load 11110101b
mov	ah,01010101b XOR 11110000b	; Load 10100101b

Relational Operators

The relational operators compare two expressions and return true (–1) if the condition specified by the operator is satisfied, or false (0) if it is not. The expressions must resolve to constant values. Relational operators are typically used with conditional directives. Table 11–3 lists the operators and the values they return if the specified condition is satisfied.

Table 11–3 . Relational Operators

Operator	Syntax	Returned Value
EQ	<i>expression1 EQ expression2</i>	True if expressions are equal
NE	<i>expression1 NE expression2</i>	True if expressions are not equal
LT	<i>expression1 LT expression2</i>	True if left expression is less than right
LE	<i>expression1 LE expression2</i>	True if left expression is less than or equal to right
GT	<i>expression1 GT expression2</i>	True if left expression is greater than right
GE	<i>expression1 GE expression2</i>	True if left expression is greater than or equal to right

Note: The EQ and NE operators treat their arguments as 16 bit numbers. Numbers specified with the 16th bit set are considered negative. For example, the expression *-1 EQ 0FFFFh* is true, but the expression *-1 NE 0FFFFh* is false.

The LT, LE, GT, and GE operators treat their arguments as 17 bit numbers, in which the 17th bit specifies the sign. For example, *0FFFFh* is 4,294,967,295, not -1. The expression *1 GT -1* is true, but the expression *1 GT 0FFFFh* is false.

```

mov     ax,4 EQ 3           ; Load false ( 0)
mov     ax,4 NE 3           ; Load true  (-1)
mov     ax,4 LT 3           ; Load false ( 0)
mov     ax,4 LE 3           ; Load false ( 0)
mov     ax,4 GT 3           ; Load true  (-1)
mov     ax,4 GE 3           ; Load true  (-1)

```

Segment Override Operators

The segment override operator (:) forces the address of a variable or label to be computed relative to a specific segment. The syntax is:

segment:expression

segment can be specified in several ways. It can be one of the segment registers: CS, DS, SS, or ES (or FS or GS on the 80386). It can also be a segment or group name. In this case, the name must have been previously defined with a SEGMENT or GROUP directive and assigned to a segment register with an ASSUME directive. The expression can be a constant, expression, or a SEG expression. See Seg Operator for more information on the SEG operator.

Note: *When a segment override is given with an indexed operand, the segment must be specified outside the index operators. For example, es:[di] is correct, but [es:di] generates an error.*

As shown in the last two statements of the following example, a segment override with a segment name is not enough if no segment register is assumed for the segment name. You must use the ASSUME statement to assign a segment register, as explained in Section 7.

```
mov      ax,ss:[bx+4]           ; Override default assume
                                   ;      (DS)
mov      al,es:082h             ; Load from ES
ASSUME   ds:FAR_DATA            ; Tell the assembler and
mov      bx,FAR_DATA:count      ;   load from a far
                                   ;   segment
```

Type Operators

This section describes the assembler operators that specify or analyze the types of memory operands and other expressions. They are as follows:

- PTR
- SHORT
- THIS
- HIGH and LOW
- SEG
- OFFSET
- .TYPE
- TYPE
- LENGTH
- SIZE

PTR Operator

The PTR operator specifies the type for a variable or label.

type PTR *expression*

The operator forces *expression* to be treated as having *type*. *expression* can be any operand. *type* can be BYTE, WORD, DWORD, FWORD, QWORD, or TBYTE for memory operands; it can be NEAR, FAR, or PROC for labels.

The PTR operator is typically used with forward references to define explicitly what size or distance a reference has. If it is not used, the assembler assumes a default size or distance for the reference. See Using Forward References for more information on forward references.

The PTR operator is also used to enable instructions to access variables in ways that would otherwise generate errors. For example, you could use the PTR operator to access the high order byte of a WORD size variable. The PTR operator is required for FAR calls and jumps to forward referenced labels.

```
stuff      .DATA
buffer     DD      ?
           DB      20 DUP (?)

           .CODE
           .
           .
call        FAR PTR task           ; Call a far
                                   ; procedure
jmp         FAR PTR place          ; Jump far
mov         bx,WORD PTR stuff[0]   ; Load a word from a
                                   ; doubleword
add         ax,WORD PTR buffer[bx] ; Add a word from a
                                   ; byte variable
```

SHORT Operator

The SHORT operator sets the type of a specified label to SHORT. Short labels can be used in JMP instructions whenever the distance from the label to the instruction is less than 128 bytes. The syntax is:

SHORT *label*

Instructions using short labels are a byte smaller than identical instructions using the default near labels. See Forward Reference to Labels, for information on using the SHORT operator with jump instructions.

```

        jmp      again                ; Jump 128 bytes or more
        .
        .
        jmp      SHORT again         ; Jump less than 128 bytes
        .
        .
again:

```

THIS Operator

The THIS operator creates an operand whose offset and segment values are equal to the current location counter value and whose type is specified by the operator. The syntax is:

THIS type

type can be BYTE, WORD, DWORD, FWORD, QWORD, or TBYTE for memory operands. It can be NEAR, FAR, or PROC for labels.

The THIS operator is typically used with the EQU or equal sign (=) directive to create labels and variables. The result is similar to using the LABEL directive.

```

tag1      EQU      THIS BYTE          ; Both represent the same variable
tag2      LABEL    BYTE

check1     EQU      THIS NEAR          ; All represent the same address
check2     LABEL    NEAR
check3:
check4     PROC      NEAR
check4     ENDP

```

HIGH and LOW Operators

The HIGH and LOW operators return the high and low bytes, respectively, of an expression. The syntax is:

HIGH expression

LOW expression

The HIGH operator returns the high order eight bits of *expression*; the LOW operator returns the low order eight bits. *expression* must evaluate to a constant. You cannot use the HIGH and LOW operators on the contents of a memory operand since the contents may change at run time.

```
stuff      EQU      0ABCDh
            mov      ah,HIGH stuff      ; Load 0ABh
            mov      al,LOW stuff       ; Load 0CDh
```

SEG Operator

The SEG operator returns the segment address of an expression. The syntax is:

SEG *expression*

expression can be any label, variable, segment name, group name, or other memory operand. The SEG operator cannot be used with constant expressions. The returned value can be used as a memory operand.

```
.          DATA
var        DB      ?
            .CODE
            .
            .
            .
            mov     ax,SEG var           ; Get address of segment
                                           ;   where variable is
                                           ;   declared
            ASSUME  ds:SEG var          ; Assume segment of
                                           ;   variable
```

OFFSET Operator

The OFFSET operator returns the offset address of an expression. The syntax is:

OFFSET *expression*

expression can be any label, variable, or other direct memory operand. Constant expressions return meaningless values. The value returned by the OFFSET operand is an immediate (constant) operand.

If simplified segment directives are given, the returned value varies. If the item is declared in a near data segment, the returned value is the number of bytes between the item and the beginning of its group (normally DGROUP). If the item is declared in a far segment, the returned value is the number of bytes between the item and the beginning of the segment.

If full segment definitions are given, the returned value is a memory operand equal to the number of bytes between the item and the beginning of the segment in which it is defined.

The segment override operator (:) can be used to force OFFSET to return the number of bytes between the item in expression and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group when full segment definitions are used. For example, the statement:

```
        mov     bx,OFFSET DGROUP:array
is not the same as:
        mov     bx,OFFSET array
if array is not the first segment in DGROUP:

string  .DATA
        DB      This is it.
        .CODE
        .
        .
        mov     dx,OFFSET string           ; Load offset of variable
```

.TYPE Operator

The .TYPE operator returns a byte that defines the mode and scope of an expression. The syntax is:

.TYPE expression

If *expression* is not valid, .TYPE returns 0. Otherwise .TYPE returns a byte having the bit setting shown in Table 11–4. Only bits 0, 1, 5, and 7 are affected. Other bits are always 0.

Table 11–4. .TYPE Operator and Variable Attributes

Bit Position	If Bit = 0	If Bit = 1
0	Not program related	Program related
1	Not data related	Data related
5	Not defined	Defined
7	Local or public scope	External scope

The .TYPE operator is typically used in macros in which different kinds of arguments may need to be handled differently.

The following macro checks to see if the argument passed to it is data related (a variable). It does this by shifting all bits except the relevant bits (1 and 0) left so that they can be checked. If the data bit is not set, an error message is generated.

```
display      MACRO      string
              IF         ((.TYPE string) SHL 14) NE 8000h
              IF2
              %OUT       Argument must be a variable
              ENDIF
              ENDIF
              push       ds
              mov        dx, OFFSET string
              push       dx
              call        zprint
              ENDM
```

TYPE Operator

The TYPE operator returns a number that represents the type of an expression. The syntax is:

TYPE expression

If *expression* evaluates to a variable, the operator returns the number of bytes in each data object in the variable. Each byte in a string is considered a separate data object, so the TYPE operator returns 1 for strings.

If *expression* evaluates to a structure or structure variable, the operator returns the number of bytes in the structure. If *expression* is a label, the operator returns 0FFFFh for NEAR labels and 0FFFEh for FAR labels. If *expression* is a constant, the operator returns 0.

The returned value can be used to specify the type for a PTR operator. For example:

```
var      .DATA
array    DW      ?
str      DD      10 DUP (?)
         DB      This is a test
         .CODE
         .
         .
         .
mov       ax,TYPE var           ; Puts 2 in AX
mov       bx,TYPE array        ; Puts 4 in BX
mov       cx,TYPE str          ; Puts 1 in CX
jmp       (TYPE room) PTR room ; Jump is near or far,
                               ; depending on
                               ; memory model
         .
room     LABEL      PROC
```

LENGTH Operator

The LENGTH operator returns the number of data elements in an array or other variable defined with the DUP operator. The syntax is:

LENGTH *variable*

The returned value is the number of elements of the declared size in *variable*. If *variable* was declared with nested DUP operators, only the value given for the outer DUP operator is returned. If *variable* was not declared with the DUP operator, the value returned is always 1.

```

array      DD      100 DUP(0FFFFFFh)table      DW      100
; DUP(1,10 DUP(?))

string     DB      'This is a string'
var        DT      ?
larray     EQU     LENGTH array                ; 100 – number of elements
ltable     EQU     LENGTH table                ; 100 – inner DUP not
; counted
lstring    EQU     LENGTH string              ; 1 – string is one
; element
lvar       EQU     LENGTH var                  ; 1
.
.
.
again:     mov     cx,LENGTH array             ; Load number of elements
; Perform some operation
; on
; each element
.
loop      again

```

SIZE Operator

The **SIZE** operator returns the total number of bytes allocated for an array or other variable defined with the **DUP** operator. The syntax is:

SIZE *variable*

The returned value is equal to the value of **LENGTH** variable times the value of **TYPE** variable. If the variable was declared with nested **DUP** operators, only the value given for the outside **DUP** operator is considered. If the variable was not declared with the **DUP** operator, the value returned is always **TYPE** variable. The following example shows several ways of using the **SIZE** operator:

```

array      DD      100 DUP(1)
table      DW      100 DUP(1,10 DUP(?))
string     DB      'This is a string'
var         DT      ?
sarray     EQU     SIZE array           ; 400 – elements times size
stable     EQU     SIZE table           ; 200 – inner DUP ignored
sstring    EQU     SIZE string          ; 1 – string is one
                                                ; element
svar       EQU     SIZE var             ; 10 – bytes in variable
.
.
.
again:     mov     cx,SIZE array         ; Load number of bytes
.                                                  ; Perform some operation
.                                                  ; on
.                                                  ; each byte
.
loop      again

```

Operator Precedence

Expressions are evaluated according to the following rules:

- Operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right.
- The order of evaluation can be overridden by using parentheses. Operations in parentheses are always performed before any adjacent operations.

The order of precedence for all operators is listed in Table 11–5. Operators on the same line have equal precedence.

Table 11–5. Operator Precedence

Precedence	Operators
(Highest)	
1	LENGTH, SIZE, WIDTH, MASK, (), [], < >
2	.(structure field name operator)
3	:
4	PTR, OFFSET, SEG, TYPE, THIS
5	HIGH, LOW
6	+ ,– (unary)
7	*,/, MOD, SHL, SHR
8	+ ,– (binary)
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13	SHORT, .TYPE
(Lowest)	

a	EQU	8 / 4 * 2	; Equals 4
b	EQU	8 / (4 * 2)	; Equals 1
c	EQU	8 + 4 * 2	; Equals 16
d	EQU	(8 + 4) * 2	; Equals 24
e	EQU	8 OR 4 AND 2	; Equals 8
f	EQU	(8 OR 4) AND 3	; Equals 0

Using the Location Counter

The location counter is a special operand that, during assembly, represents the address of the statement currently being assembled. At assembly time, the location counter keeps changing, but when used in source code it resolves to a constant representing an address.

The location counter has the same attributes as a near label. It represents an offset that is relative to the current segment and is equal to the number of bytes generated for the segment to that point.

The following example shows one way of using the location counter operand in expressions relating to data.

```
string      DB      Who wants to count every byte in a string,  
            DB      especially if you might change it later.  
lstring     EQU     $-string ; Let the assembler do it
```

The following example illustrates how you can use the location counter to do conditional jumps of more than 128 bytes. The first part shows the normal way of coding jumps of less than 128 bytes, and the second part shows how to code the same jump when the label is more than 128 bytes away.

```
            cmp      ax,bx  
            jl       shortjump      ; If ax < bx, go to shortjump  
            .         ;     else if ax >= bx, continue  
            .  
shortjump:  .  
  
            cmp      ax,bx  
            jge      $+5             ; If ax >= bx, continue  
            jmp      longjump       ;     else if ax < bx, go to  
            .         ;     longjump  
            .         ; This is $+5  
longjump:  .  
            .
```

Using Forward References

The assembler permits you to refer to labels, variable names, segment names, and other symbols before they are declared in the source code. Such references are called forward references.

The assembler handles forward references by making assumptions about them on the first pass and then attempting to correct the assumptions, if necessary, on the second pass. Checking and correcting assumptions on the second pass takes processing time, so source code with forward references assembles more slowly than source code with no forward references.

In addition, the assembler may make incorrect assumptions that it cannot correct, or corrects at a cost in program efficiency.

The subsections explain the following usage of forward references:

- Forward References to Labels
- Forward References to Variables

Forward References to Labels

Forward references to labels can result in incorrect or inefficient code.

In the following statement, the label `target` is a forward reference:

```
        jmp          target          ; Generates 3 bytes  
        .            ;      in 16 bit segment  
        .  
        .  
target:
```

Since the assembler processes source files sequentially, `target` is unknown when it is first encountered. Assuming 16 bit segments, it could be one of three types: short (–128 to 127 bytes from the jump), near (–32,768 to 32,767 bytes from the jump), or far (in a different segment than the jump). MASM assumes that `target` is a near label, and assembles the number of bytes necessary to specify a near label: one byte for the instruction and two bytes for the operand.

If on the second pass, the assembler learns that target is a short label, it needs only two bytes: one for the instruction and one for the operand. However, it is not able to change its previous assembly and the three byte version of the assembly will stand. If the assembler learns that target is a far label, it needs five bytes. Since it can't make this adjustment, it generates a phase error.

You can override the assembler's assumptions by specifying the exact size of the jump. For example, if you know that a JMP instruction refers to a label less than 128 bytes from the jump, you can use the SHORT operator, as shown:

```
        jmp          SHORT target          ; Generates 2 bytes
        .
        .
        .
target:
```

Using the SHORT operator makes the code smaller and slightly faster. If the assembler has to use the three byte form when the two byte form would be acceptable, it will generate a warning message if the warning level is 2. (The warning level can be set with the /W option, as described in Section 4.) You can ignore the warning, or you can go back to the source code and change the code to eliminate the forward references.

Note: *The SHORT operator in this example above would not be needed if target were located before the jump. The assembler would have already processed target and would be able to make adjustments based on its distance.*

If you use the SHORT operator when the label being jumped to is more than 128 bytes away, MASM generates an error message. You can either remove the SHORT operator, or try to reorganize your program to reduce the distance.

If a far jump to a forward referenced label is required, you must override the assembler's assumptions with the FAR and PTR operators, as shown:

```
        jmp          FAR PTR target        ; Generates 5 bytes
        .
        .
        .
target:                                ; In different segment
```

If the type of a label has been established earlier in the source code with an EXTRN directive, the type does not need to be specified in the jump statement.

If the 80386 processor is enabled, jumps with forward references have different limitations. One difference is that conditional jumps can be either short or near. With previous processors, all conditional jumps were short. For 32 bit segments, the number of bytes generated for near and far jumps is greater in order to handle the larger addresses in the operand.

.MODEL	large	; Model comes first, so
		; use
.386		; 16 bit segments
.CODE		
.		
.		
jmp	SHORT place	; Short unconditional jump
		; -2 bytes
jne	SHORT place	; Short conditional jump
		; -2 bytes
jmp	place	; Near unconditional jump
		; -3 bytes
jne	place	; Near conditional jump
		; -4 bytes
jmp	FAR PTR place	; Far unconditional jump
		; -5 bytes
.386		; .386 comes first, so use
.MODEL	large	; 32 bit segments
.CODE		
.		
.		
jmp	SHORT place	; Short unconditional jump
		; -2 bytes
jne	SHORT place	; Short conditional jump
		; -2 bytes
jmp	place	; Near unconditional jump
		; -5 bytes
jne	place	; Near conditional jump
		; -6 bytes
jmp	FAR PTR place	; Far unconditional jump
		; -7 bytes

Forward References to Variables

When MASM encounters code referencing variables that have not yet been defined in Pass 1, it makes assumptions about the segment where the variable will be defined. If on Pass 2 the assumptions turn out to be wrong, an error will occur.

These problems usually occur with complex segment structures that do not follow the Microsoft segment conventions. The problems never appear if simplified segment directives are used.

By default, MASM assumes that variables are referenced to the DS register. If a statement must access a variable in a segment not associated with the DS register, and if the variable has not been defined earlier in the source code, you must use the segment override operator to specify the segment.

The situation is different if neither the variable nor the segment in which it is defined has been defined earlier in the source code. In this case, you must assign the segment to a group earlier in the source code. MASM will then know about the existence of the segment even though it has not yet been defined.

Strong Typing for Memory Operands

The assembler carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that when an instruction uses two operands with implied data types, the operand types must match. Warning messages are generated for nonmatching types.

For example, in the following fragment, the variable `string` is incorrectly used in a move instruction:

```
string      .DATA
            DB          A message.
            .CODE
            .
            .
            mov         ax,string[1]
```

The AX register has WORD type, but string has BYTE type. Therefore, the statement generates warning message 37:

Operand types must match

To avoid all ambiguity and prevent the warning error, use the PTR operator to override the variable's type, as shown below:

```
mov ax,WORD PTR string[1]
```

You can ignore the warnings if you are willing to trust the assembler's assumptions. When a register and memory operand are mixed, the assembler assumes that the register operand is always the correct size. For example, in the statement

```
mov ax,string[1]
```

the assembler assumes that the programmer wishes the word size of the register to override the byte size of the variable. A word starting at string[1] will be moved into AX. In the statement

```
mov string[1],ax
```

the assembler assumes that the programmer wishes to move the word value in AX into the word starting at string[1]. However, the assembler's assumptions are not always as clear as in these examples. You should not ignore warnings about type mismatches unless you are sure you understand how your code will be assembled.

Note: *Some assemblers (including early versions of the IBM Macro Assembler) do not do strict type checking. For compatibility with these assemblers, type errors are warnings rather than severe errors. Many assembly language program listings in books and magazines are written for assemblers with weak type checking. Such programs may produce warning messages, but assemble correctly. You can use the /W option to turn off type warnings if you are sure the code is correct.*

Section 12

Assembling Conditionally

The Macro Assembler provides two types of conditional directives:

- Conditional assembly directives
- Conditional error directives

Conditional assembly directives test for a specified condition and assemble a block of statements if the condition is true. Conditional error directives test for a specified condition and generate an assembly error if the condition is true.

Both kinds of conditional directives test assembly time conditions. They cannot test run time conditions. Only expressions that evaluate to constants during assembly can be compared or tested.

Since macros and conditional assembly directives are often used together, you may need to refer to Section 13 to understand some of the examples in this section. In particular, conditional directives are frequently used with the operators described in Section 13.

Using Conditional Assembly Directives

The conditional assembly directives include the following:

IF	IFDEF	IFNB
IF1	IFDIF	IFNDEF
IF2	IFE	ENDIF
IFB	IFIDN	ELSE

The IF directives and the ENDIF and ELSE directives can be used to enclose the statements to be considered for conditional assembly. The syntax is:

```
IF condition
statements
[[ELSE
statements ]]
ENDIF
```

statements following the IF directive can be any valid statements, including other conditional blocks. The ELSE directive and its *statements* are optional. ENDIF ends the block.

The statements in the conditional block are assembled only if *condition* specified by the corresponding IF statement is satisfied. If the conditional block contains an ELSE directive, only the statements up to the ELSE directive are assembled. The statements that follow the ELSE directive are assembled only if the IF statement is not met. An ENDIF directive must mark the end of any conditional assembly block. No more than one ELSE directive is allowed for each IF statement.

IF statements can be nested up to 255 levels. A nested ELSE directive always belongs to the nearest preceding IF statement that does not have its own ELSE

You can use conditional assembly directives to:

- Testing Expressions with IF and IFE Directives
- Testing the Pass with IF1 and IF2 Directives
- Testing Symbol Definition with IFDEF and IFNDEF Directives
- Verifying Macro Parameters with IFB and IFNB Directives
- Comparing Macro Arguments with IFIDN and IFDIF Directives

Testing Expressions with IF and IFE Directives

The IF and IFE directives test the value of an expression and grant assembly based on the result. The syntax is:

IF expression

IFE expression

The IF directive grants assembly if the value of *expression* is true (nonzero). The IFE directive grants assembly if the value of *expression* is false (0). The expression must resolve to a constant value and must not contain forward references.

In the following example, a different debug routine will be called, depending on the value of debug

```
IF          debug GT 20
push       debug
call       adebug
ELSE
call       bdebug
ENDIF
```

Testing the Pass with IF1 and IF2 Directives

The IF1 and IF2 directives test the current assembly pass and grant assembly on the pass specified by the directive. Multiple passes of the assembler are discussed in Section 4. The syntax is:

IF1

IF2

The IF1 directive grants assembly only on Pass 1. IF2 grants assembly only on Pass 2. The directives take no arguments.

Macros usually only need to be processed once. You can enclose blocks of macros in IF1 blocks to prevent them from being reprocessed on the second pass.

```
dostuff    IF1
            MACRO          argument          ; Define on first pass only
            .
            .
            .
            ENDM
            ENDIF
```

Testing Symbol Definition with IFDEF and IFNDEF Directives

The IFDEF and IFNDEF directives test whether or not a symbol has been defined and grant assembly based on the result. The syntax is:

IFDEF *name*

IFNDEF *name*

The IFDEF directive grants assembly only if *name* is a defined label, variable, or symbol. The IFNDEF directive grants assembly if *name* has not yet been defined.

name can be any valid name. Note that if *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

In the following example, buff is allocated only if buffer has been previously defined.

```
buff          IFDEF          buffer
               DB             buffer DUP(?)
               ENDEF
```

One way to use this conditional block is to leave buffer undefined in the source file and define it if needed by using the /D symbol option (see Section 4) when you start MASM. For example, if the conditional block is in test.asm, you could start the assembler with the following command line:

```
MASM
[Args] /Dbuffer=1024 test;
```

The command line would define the symbol buffer; as a result, the conditional assemble would allocate buff. However, if you didn't need buff, you could use the following command line:

```
MASM
[Args] test;
```

Verifying Macro Parameters with IFB and IFNB Directives

The IFB and IFNB directives test to see if a specified argument was passed to a macro and grant assembly based on the result.

IFB <*argument*>

IFNB <*argument*>

These directives are always used inside macros, and they always test whether a real argument was passed for a specified dummy argument. The

IFB directive grants assembly if argument is blank. The IFNB directive grants assembly if argument is not blank. The arguments can be any name, number, or expression. Angle brackets (< >) are required.

In the following example, a default value is used if no value is specified for the third macro argument.

```

Write      MACRO      buffer,bytes,handle
           IFNB      <handle>
           mov       dx,OFFSET handle      ;
           ELSE
           mov       dx,OFFSET bsVid1      ; Default standard out
           ENDF
           mov       dx,OFFSET buffer      ; Address of buffer to
           push     ds
           push     dx
           push     ds
           push     OFFSET buffer
           push     bytes
           push     ds
           push     OFFSET cbwritten
           call     WriteBSRecord
           ENDM

```

Comparing Macro Arguments with IFIDN and IFDIF Directives

The IFIDN and IFDIF directives compare two macro arguments and grant assembly based on the result.

IFIDN **[I]** <*argument1*>,<*argument2*>

IFDIF **[I]** <*argument1*>,<*argument2*>

These directives are always used inside macros, and they always test whether real arguments passed for two specified arguments are the same. The IFIDN directive grants assembly if *argument1* and *argument2* are identical. The IFDIF directive grants assembly if *argument1* and *argument2* are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional I at the end of the directive name specifies that the directive is case insensitive. Arguments that are spelled the same are evaluated the same, regardless of case. If the I is not given, the directive is case sensitive.


```
divide8      MACRO      numerator,denominator
AL           IFDIFI      <numerator>,<al>      ;; If numerator isn't
                                     mov      al,numerator      ;; make it AL
                                     ENDF
                                     xor      ah,ah
                                     div      denominator
                                     ENDM
```

In this example, a macro uses the IFDIFI directive to check one of the arguments and take a different action, depending on the text of the string. The sample macro can be enhanced further by checking for other values that require adjustment (such as a denominator passed in AL or passed in AH).

Using Conditional Error Directives

This subsection explains how to use directives to do the following:

- Generate Unconditional Errors
- Test Expressions
- Verify Symbol Definitions
- Test for Macro Parameters
- Compare Macro Arguments

Conditional error directives can be used to debug programs and check for assembly time errors. By inserting a conditional error directive at a key point in your code, you can test assembly time conditions at that point. You can also use conditional error directives to test for boundary conditions in macros.

The conditional error directives and the error messages they produce are listed in Table 12–1.

Table 12–1. Conditional Error Directives

Directive	Number	Message
.ERR	187	Forced error – pass1
.ERR2	88	Forced error – pass2
.ERR	89	Forced error
.ERRE	90	Forced error – expression true (0)
.ERRNZ	91	Forced error – expression false (not 0)
.ERRNDEF	92	Forced error – symbol not defined
.ERRDEF	93	Forced error – symbol defined
.ERRB	94	Forced error – string blank
.ERRNB	95	Forced error – string not blank
.ERRIDN[I]	96	Forced error – strings identical
.ERRDIF[I]	97	Forced error – strings different

Like other severe errors, those generated by conditional error directives cause the assembler to return exit code 7. If a severe error is encountered during assembly, MASM deletes the object module. All conditional error directives except `ERR1` generate severe errors.

Generating Unconditional Errors with `.ERR`, `.ERR1`, and `.ERR2` Directives

The `.ERR`, `.ERR1`, and `.ERR2` directives force an error where the directives occur in the source file. The error is generated unconditionally when the directive is encountered, but the directives can be placed within conditional assembly blocks to limit the errors to certain situations. The syntax is:

```
.ERR
.ERR1
.ERR2
```

The `.ERR` directive forces an error regardless of the pass. The `.ERR1` and `.ERR2` directives force the error only on their respective passes. The `.ERR1` directive appears only on the screen or in the listing file if you use the `/D` option to request a Pass 1 listing.

You can place these directives within conditional assembly blocks or macros to see which blocks are being expanded.

```
IFDEF      dos
.
.
.
ELSE
IFDEF      xenix
.
.
.
ELSE
.ERR
OUT dos or xenix must be defined
ENDIF
ENDIF
```

This example makes sure that either the symbol `dos` or the symbol `xenix` is defined. If neither is defined, the nested `ELSE` condition is assembled and an error message is generated. Since the `.ERR` directive is used, an error would be generated on each pass. You could use `.ERR1` or `.ERR2` to check if you want the error to be generated only on the corresponding pass.

Testing Expressions with `.ERRE` or `.ERRNZ` Directives

The `.ERRE` and `.ERRNZ` directives test the value of an expression and conditionally generate an error based on the result. The syntax is:

```
.ERRE expression
.ERRNZ expression
```

The `.ERRE` directive generates an error if *expression* is false (0). The `.ERRNZ` directive generates an error if *expression* is true (nonzero). The expression must resolve to a constant value and must not contain forward references.

In the following example, the `.ERRE` directive is used to check the boundaries of a parameter passed to the macro buffer. If `count` is less than or equal to 128, the expression being tested by the error directive will be true (nonzero) and no error will be generated. If `count` is greater than 128, the expression will be false (0) and the error will be generated.

```
buffer      MACRO    count,bname
             .ERRE    count LE 128                ;; Allocate memory, but
bname       DB        count DUP(0)                ;; no more than 128
                                                    ;;   bytes
             ENDM
             .
             .
             buffer   128,buf1                    ; Data allocated – no
                                                    ;   error
             buffer   129,buf2                    ; Error generated
```

Verifying Symbol Definitions with .ERRDEF and .ERRNDEF Directives

The .ERRDEF and .ERRNDEF directives test whether or not a symbol is defined and conditionally generate an error based on the result. The syntax is:

```
.ERRDEF name
.ERRNDEF name
```

The .ERRDEF directive produces an error if *name* is defined as a label, variable, or symbol. The .ERRNDEF directive produces an error if *name* has not yet been defined. If *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

In the following example, the .ERRNDEF directive at the beginning of the conditional block makes sure that a symbol being tested in the block actually exists.

```
.ERRNDEF    publevel
IF          publevel LE 2
PUBLIC      var1, var2
ELSE
PUBLIC      var1, var2, var3
ENDIF
```

Testing for Macro Parameters with .ERRB and .ERRNB Directives

The .ERRB and .ERRNB directives test whether a specified argument was passed to a macro and conditionally generate an error based on the result. The syntax is:

```
.ERRB <argument>  
.ERRNB <argument>
```

These directives are always used inside macros, and they always test whether a real argument was passed for a specified dummy argument. The directive generates an error if *argument* is blank. The .ERRNB directive generates an error if *argument* is not blank. The argument can be any name, number, or expression. Angle brackets (< >) are required.

In the following example, error directives are used to make sure that one, and only one, argument is passed to the macro. The .ERRB directive generates an error if no argument is passed to the macro. The .ERRNB directive generates an error if more than one argument is passed to the macro.

```
work          MACRO realarg,testarg  
               .ERRB <realarg>    ;; Error if no parameters  
               .ERRNB <testarg>   ;; Error if more than one  
               ;;      parameter  
               .  
               .  
               .  
            ENDM
```

Comparing Macro Arguments with .ERRIDN and .ERRDIF Directives

The .ERRIDN and .ERRDIF directives compare two macro arguments and conditionally generate an error based on the result. The syntax is:

```
.ERRIDN [I] <argument1>,<argument2>  
.ERRDIF [I] <argument1>,<argument2>
```

These directives are always used inside macros, and they always compare the real arguments specified for two parameters. The `.ERRIDN` directive generates an error if the arguments are identical. The `.ERRDIF` directive generates an error if the arguments are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional `I` at the end of the directive name specifies that the directive is case insensitive. Arguments that are spelled the same are evaluated the same regardless of case. If the `I` is not given, the directive is case sensitive.

In the following example, the `.ERRIDNI` directive is used to protect against passing the `AX` register as the second parameter, since this would cause the macro to fail.

```
addem          MACRO      ad1,ad2,sum
                .ERRIDNI   <ax>,<ad2> ;; Error if ad2 is ax
                mov        ax,ad1      ;; Would overwrite if ad2 were AX
                add        ax,ad2
                mov        sum,ax      ;; Sum must be register or memory
                ENDM
```


Section 13

Using Equates, Macros, and Repeat Blocks

Equates are constant values assigned to symbols so that the symbol can be used in place of the value. Macros are a series of statements that are assigned a symbolic name (and optionally parameters) so that the symbol can be used in place of the statements. Repeat blocks are a special form of macro used to do repeated statements.

Both equates and macros are processed at assembly time. They can simplify writing source code by allowing the user to substitute mnemonic names for constants and repetitive code. By changing a macro or equate, you can change the effect of statements throughout the source code.

In exchange for these conveniences, you lose some assembly time efficiency. Assembly may be slightly slower for a program that uses macros and equates extensively than for the same program written without them. However, the program without macros and equates usually takes longer to write and is more difficult to maintain.

The following subsections explain how to:

- Use equates
- Use macros
- Define repeat blocks
- Use macro operators
- Use recursive, nested, and redefined macros
- Manage macros and equates

Using Equates

The equate directives enable you to use symbols that represent numeric or string constants. MASM recognizes three kinds of equates:

- Redefinable numeric equates
- Nonredefinable numeric equates
- String equates (also called text macros)

Redefinable Numeric Equates

Redefinable numeric equates are used to assign a numeric constant to a symbol. The value of the symbol can be redefined at any point during assembly time. Although the value of a redefinable equate may be different at different points in the source code, a constant value will be assigned for each use, and that value will not change at run time.

Redefinable equates are often used for assembly time calculations in macros and repeat blocks. The syntax is:

name = *expression*

The equal sign (=) directive creates or redefines a constant symbol by assigning the numeric value of *expression* to *name*. No storage is allocated for the symbol. The symbol can be used in subsequent statements as an immediate operand having the assigned value. It can be redefined at any time.

The *expression* can be an integer, a constant expression, a one or two character string constant (four character on the 80386), or an expression that evaluates to an address. The name must be either a unique name or a name previously defined by using the equal sign (=) directive.

Note: *Redefinable equates must be assigned numeric values. String constants longer than two characters cannot be used.*

```
counter    =      0          ; Initialize counterarray
           LABEL  BYTE      ; Label array of
                           ;   increasing numbers
           REPT   100        ; Repeat 100 times
           DB     counter    ; Initialize number
counter    =      counter + 1 ; Increment counter
           ENDM
```

This example redefines equates inside a repeat block to declare an array initialized to increasing values from 0 to 100. The equal sign directive is used to increment the counter symbol for each loop. See Defining Repeat Blocks for more information.

Nonredefinable Numeric Equates

Nonredefinable numeric equates are used to assign a numeric constant to a symbol. The value of the symbol cannot be redefined.

Nonredefinable numeric equates are often used for assigning mnemonic names to constant values. This can make the code more readable and easier to maintain. If a constant value used in numerous places in the source code needs to be changed, then the equate can be changed in one place rather than throughout the source code. The syntax is:

name EQU expression

The EQU directive creates constant symbols by assigning *expression* to *name*. The assembler replaces each subsequent occurrence of *name* with the value of *expression*. Once a numeric equate has been defined with the EQU directive, it cannot be redefined. Attempting to do so generates an error.

Note: *String constants can also be defined with the EQU directive, but the syntax is different, as described in String Equates.*

No storage is allocated for the symbol. Symbols defined with numeric values can be used in subsequent statements as immediate operands having the assigned value. For example:

```
column    EQU    80           ; Numeric constant 80
row       EQU    25           ; Numeric constant 25
screenful EQU    column * row ; Numeric constant 2000
line      EQU    row          ; Alias for row

        .DATA
buffer   DW      screenful

        .CODE
        .
        .
        .
        mov     cx,column
        mov     bx,line
```

String Equates

String equates (or text macros) are used to assign a string constant to a symbol. String equates can be used in a variety of contexts, including defining aliases and string constants.

name *EQU* [*<*]string[*>*]

The EQU directive creates constant symbols by assigning *string* to *name*. The assembler replaces each subsequent occurrence of *name* with *string*. Symbols defined to represent strings with the EQU directive can be redefined to new strings. Symbols cannot be defined to represent strings with the equal sign (=) directive.

An alias is a special kind of string equate. It is a symbol that is equated to another symbol or keyword.

The use of angle brackets (< >) to force string evaluation is a new feature of Version 5.0 of the Macro Assembler. Previous versions tried to evaluate equates as expressions. If the string did not evaluate to a valid expression, MASM evaluated it as a string. This behavior sometimes caused unexpected consequences.

For example, the statement

```
rt      EQU      runtime
```

would be evaluated as run minus time, even though the user might intend to define the string runtime. If run and time were not already defined as numeric equates, the statement would generate an error. Using angle brackets solves this problem. The statement:

```
rt      EQU      <runtime>
```

is evaluated as the string runtime

When maintaining existing source code, you can leave string equates alone that evaluate correctly, but for new source code that will not be used with previous versions of MASM, it is a good idea to enclose all string equates in angle brackets.

```

; String equate definitions
pi      EQU      <3.1415>      ; String constant 3.1415
prompt  EQU      <'Type Name: ' > ; String constant 'Type
                                           ; Name: '
WPT      EQU      <WORD PTR>    ; String constant for WORD
                                           ; PTR
arg1     EQU      <[bp+4]>       ; String constant for
                                           ; [bp+4]

; Use of string equates
        .DATA
message DB      prompt          ; Allocate string Type
                                           ; Name:
pie      DQ      pi              ; Allocate real number
                                           ; 3.1415

        .CODE
        .
        .
        .
        inc     WPT parm1       ; Increment word value of
                                           ; argument passed on
                                           ; stack

```

Using Macros

Macros enable you to assign a symbolic name to a block of source statements, and then to use that name in your source file to represent the statements. Parameters can also be defined to represent arguments passed to the macro. The subsections listed below explain how to use macros:

- Defining Macros
- Calling Macros
- Using Local Symbols
- Exiting from a Macro

Macro expansion is a text processing function that occurs at assembly time. Each time MASM encounters the text associated with a macro name, it replaces that text with the text of the statements in the macro definition. Similarly, the text of parameter names is replaced with the text of the corresponding actual arguments.

A macro can be defined any place in the source file as long as the definition precedes the first source line that calls the macro. Macros and equates are often kept in a separate file and made available to the program through an `INCLUDE` directive (For more information see Using Include Files) at the start of the source code.

Note: *Since most macros only need to be expanded once, you can increase efficiency by processing them only during a single pass of the assembler. You can do this by enclosing the macros (or an `INCLUDE` statement that calls them) in a conditional block using the `IF1` directive.*

Often a task can be done by using either a macro or procedure. For example, the `addup` procedure shown in Section 19 does the same thing as the `addup` macro in Defining Macros. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly. Procedures are coded only once in the executable file, but the increased overhead of saving and restoring addresses and parameters can make them slower.

The section below tells how to define and call macros. Repeat blocks, a special form of macro for doing repeated operations, are discussed separately in Defining Repeat Blocks.

Defining Macros

The `MACRO` and `ENDM` directives are used to define macros. `MACRO` designates the beginning of the macro block and `ENDM` designates the end of the macro block. The syntax is:

```
name MACRO [[parameter [[parameter ]... ]  
statements  
ENDM
```

name must be unique and a valid symbol name. It can be used later in the source file to invoke the macro.

The parameters (sometimes called dummy parameters) are names that act as placeholders for values to be passed as arguments to the macro when it is called. Any number of parameters can be specified, but they must all fit on one line. If you give more than one parameter, you must separate them with commas, spaces, or tabs. Commas can always be used as separators; spaces and tabs may cause ambiguity if the arguments are expressions.

Note: *This manual uses the term parameter to refer to a placeholder for a value that will be passed to a macro or procedure. Parameters appear in macro or procedure definitions. The term argument is used to refer to an actual value passed to the macro or procedure when it is called.*

Any valid assembler statement may be placed within a macro, including statements that call or define other macros. Any number of statements can be used. The parameters can be used any number of times in the statements. Macros can be nested, redefined, or used recursively, as explained in Using Recursive, Nested, and Redefined Macros.

MASM assembles the statements in a macro only if the macro is called, and only at the point in the source file from which it is called. The macro definition itself is never assembled.

A macro definition can include the **LOCAL** directive, which lets you define labels used only within a macro, or the **EXITM** directive, which allows you to exit from a macro before all the statements in the block are expanded. These directives are discussed in Using Local Symbols, and Exiting from a Macro. Macro operators can also be used in macro definitions, as described in Using Macro Operators.

The following example defines a macro named **addup**, which uses three parameters to add three values and leave their sum in the **AX** register. The three parameters will be replaced with arguments when the macro is called.

```
addup    MACRO    ad1,ad2,ad3
          mov      ax,ad1          ;; First parameter in AX
          add      ax,ad2          ;; Add next two parameters
          add      ax,ad3          ;; and leave sum in AX
          ENDM
```

Calling Macros

A macro call directs MASM to copy the statements of the macro to the point of the call and to replace any parameters in the macro statements with the corresponding actual arguments.

name [[*argument* [,*argument*]...]

name must be the name of a macro defined earlier in the source file. The arguments can be any text. For example, symbols, constants, and registers are often given as arguments. Any number of arguments can be given, but they must all fit on one line. Multiple arguments must be separated by commas, spaces, or tabs.

MASM replaces the first parameter with the first argument, the second parameter with the second argument, and so on. If a macro call has more arguments than the macro has parameters, the extra arguments are ignored. If a call has fewer arguments than the macro has parameters, any remaining parameters are replaced with a null (empty) string.

You can use conditional statements to enable macros to check for null strings or other types of arguments. The macro can then take appropriate action to adjust to different kinds of arguments. See Section 12 for more information on using conditional assembly and conditional error directives to test macro arguments.

When the following macro is called, MASM replaces the parameters with the actual parameters given in the macro call.

```
addup      MACRO      ad1,ad2,ad3      ;; Macro definition
           mov        ax,ad1           ;; First parameter in AX
           add        ax,ad2           ;; Add next two parameters
           add        ax,ad3           ;; and leave sum in AX
           ENDM
           .
           .
           .
           addup      bx,2,count      ; Macro call
```

When expanding `addup`, the assembler generates the following code:

```
mov        ax,bx
add        ax,2
add        ax,count
```

This code could be shown in an assembler listing, depending on whether the `.LALL`, `.XALL`, or `.SALL` directive was in effect.

Using Local Symbols

The LOCAL directive can be used within a macro to define symbols that are available only within the defined macro.

Note: *In this context, the term local is not related to the public availability of a symbol, as described in Section 10 or to variables that are defined to be local to a procedure, as described in Section 19. Local simply means that the symbol is not known outside the macro where it is defined. The syntax is:*

LOCAL *localname*[[*localname*]].

localname is a temporary symbol name that is to be replaced by a unique symbol name when the macro is expanded. At least one *localname* is required for each LOCAL directive. If more than one local symbol is given, the names must be separated with commas. Once declared, *localname* can be used in any statement within the macro definition.

MASM creates a new actual name for *localname* each time the macro is expanded. The actual name has the following form:

??*number*

number is a hexadecimal number in the range 0000 to 0FFFFF. You should not give other symbols names in this format, since doing so may produce a symbol with multiple definitions. In listings, the local name is shown in the macro definition, but the actual name is shown in expansions of macro calls.

Nonlocal labels may be used in a macro. If the macro is used more than once, the same label will appear in both expansions, and MASM will display an error message, indicating that the file contains a symbol with multiple definitions. To avoid this problem, use only local labels (or redefinable equates) in macros.

Note: *The LOCAL directive can only be used in macro definitions, and it must precede all other statements in the definition. If you try another statement (such as a comment instruction) before the LOCAL directive, an error is generated.*

In the following example, the LOCAL directive defines the local names again and gotzero as labels to be used within the power macro.

```
power    MACRO    factor,exponent    ;; Use for unsigned only
        LOCAL    again,gotzero      ;; Declare symbols for
        ;;      macro
        xor       dx,dx              ;; Clear DX
        mov      cx,exponent         ;; Exponent is count for
        ;;      loop
        mov      ax,1                ;; Multiply by 1 first
        ;;      time
        jcxz     gotzero             ;; Get out if exponent is
        ;;      zero
again:    mov      bx,factor
        mul      bx                  ;; Multiply until done
        loop     again
gotzero:
        ENDM
```

These local names will be replaced with unique names each time the macro is expanded. For example, the first time the macro is called, again will be assigned the name ??000 0 and gotzero will be assigned ??0001 The second time through, again will be assigned ??0002 and gotzero will be assigned ??0003, and so on.

Exiting from a Macro

Normally, MASM processes all the statements in a macro definition and then continues with the next statement after the macro call. However, you can use the EXITM directive to tell the assembler to terminate macro expansion before all the statements in the macro have been assembled.

When the EXITM directive is encountered, the assembler exits the macro or repeat block immediately. Any remaining statements in the macro or repeat block are not processed. If EXITM is encountered in a nested macro or repeat block, MASM returns to expanding the outer block.

The EXITM directive is typically used with conditional directives to skip the last statements in a macro under specified conditions. Often macros using the EXITM directive contain repeat blocks or are called recursively.

The following example defines a macro that allocates a variable amount of data, but no more than 255 bytes. The macro contains an IFE directive that checks the expression $x - 0FFh$. When the value of this expression is true ($x - 255 = 0$), the EXITM directive is processed and expansion of the macro stops.

```
allocate    MACRO    times    ;; Macro definition
x           =        0
           REPT      times    ;; Repeat up to 256 times
           IF        x GT 0FFh ;; Is x > 255 yet?
           EXITM      ;; If so, quit
           ELSE
           DB         x        ;; Else allocate x
ENDIF
x           =        x + 1    ;; Increment x
           ENDM
           ENDM
```

Defining Repeat Blocks

Repeat blocks are a special form of macro that allows you to create blocks of repeated statements. Three different kinds of repeat blocks can be defined by using one of the three directives defined as follows:

- The REPT directive
- The IRP directive
- The IRPC directive

The difference between them is in how the number of repetitions is specified.

Repeat blocks differ from macros in that they are not named, and thus cannot be called. However, like macros, they can have parameters that are replaced by actual arguments during assembly. Macro operators, symbols declared with the LOCAL directive, and the EXITM directive can be used in repeat blocks. Like macros, repeat blocks are always terminated by an ENDM directive.

Repeat blocks are frequently placed in macros in order to repeat some of the statements in the macro. They can also be used independently, usually for declaring arrays with repeated data elements.

Repeat blocks are processed at assembly time and should not be confused with the REP instruction, which causes string instructions to be repeated at run time.

The REPT Directive

The REPT directive is used to create repeat blocks in which the number of repetitions is specified with a numeric argument.

```
REPT expression
statements
ENDM
```

expression must evaluate to a numeric constant (a 16 bit unsigned number). It specifies the number of repetitions. Any valid assembler statements may be placed within the repeat block.

The following example repeats the equal sign (=) and DB directives to initialize ASCII values for each uppercase letter of the alphabet.

```
alphabet LABEL BYTE
x          = 0           ;; Initialize
          REPT 26        ;; Specify 26 repetitions
          DB 'A' + x     ;; Allocate ASCII code for letter
x          = x + 1       ;; Increment
          ENDM
```

The IRP Directive

The IRP directive is used to create repeat blocks in which the number of repetitions, as well as parameters for each repetition, are specified in a list of arguments. The syntax is:

```
IRP parameter,argument [,argument] ...
statements
ENDM
```

The assembler statements inside the block are repeated once for each argument in the list enclosed by angle brackets (< >). The parameter is a name for a placeholder to be replaced by the current argument. Each argument can be text, such as a symbol, string, or numeric constant. Any number of arguments can be given. If multiple arguments are given, they must be separated by commas. The angle brackets (< >) around the argument list are required. *parameter* can be used any number of times in the statements.

When MASM encounters an IRP directive, it makes one copy of the statements for each argument in the enclosed list. While copying the statements, it substitutes the current argument for all occurrences of parameter in these statements. If a null argument (< >) is found in the list, the dummy name is replaced with a null value. If the argument list is empty, the IRP directive is ignored and no statements are copied.

The following example repeats the DB directive 10 times, allocating 10 bytes for each number in the list. The resulting statements create 100 bytes of data, starting with 10 zeros, followed by 10 ones, and so on.

```
numbers    LABEL    BYTE
            IRP      x, <0,1,2,3,4,5,6,7,8,9>
            DB       10 DUP(x)
            ENDM
```

The IRPC Directive

The IRPC directive is used to create repeat blocks in which the number of repetitions, as well as arguments for each repetition, is specified in a string. The syntax is:

```
IRPC parameter,string
    statements
ENDM
```

The assembler statements inside the block are repeated as many times as there are characters in *string*. *parameter* is a name for a placeholder to be replaced by the current character in *string*. *string* can be any combination of letters, digits, and other characters. It should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters. *parameter* can be used any number of times in these statements.

When MASM encounters an IRPC directive, it makes one copy of the statements for each character in the string. While copying the statements, it substitutes the current character for all occurrences of parameter in these statements.

Using Equates, Macros, and Repeat Blocks

The following example repeats the DB directive 10 times, once for each character in the string 0123456789. The resulting statements create 10 bytes of data having the values 0 through 9.

```
ten LABEL BYTE
      IRPC x,0123456789
      DB x
      ENDM
```

The following example allocates the ASCII codes for uppercase, lowercase, and numeric versions of each letter in the string. Notice that the substitute operator (&) is required so that letter will be treated as an argument rather than a string.

```
IRPC letter,ABCDEFGHIJKLMNOPQRSTUVWXYZ
DB '&letter' ; Allocate uppercase
; letter
DB '&letter'+20h ; Allocate lowercase
; letter
DB '&letter'-40h ; Allocate number of
; letter
ENDM
```

Using Macro Operators

Macro and conditional directives use the following special set of macro operators:

Table 13–1. Macro Operators

Operator	Definition
&	Substitute operator
< >	Literal text operator
!	Literal character operator
%	Expression operator
::	Macro comment

When used in a macro definition, a macro call, a repeat block, or as the argument of a conditional assembly directive, these operators carry out special control operations, such as text substitution.

Substitute Operator

The substitute operator (&) forces MASM to replace a parameter with its corresponding actual argument value. The syntax is:

¶meter

The substitute operator can be used when a parameter immediately precedes or follows other characters, or whenever the parameter appears in a quoted string.

In the following example, MASM replaces &x with the value of the argument passed to the macro errgen.

```
errgen      MACRO      y,x
            PUBLIC     err&y
err&y       DB          'Error &y: &x'
            ENDM
```

If the macro is called with the statement:

```
errgen      5,<Unreadable disk
```

the macro is expanded to:

```
err5        DB          'Error 5: Unreadable disk'
```

For complex, nested macros, you can use extra ampersands (&) to delay the replacement of a parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with z to make sure its replacement occurs while the IRP directive is being processed:

```
alloc       MACRO      x
            IRP         z,<1,2,3>
x&&z        DB          z
            ENDM
            ENDM
```

In this example, the dummy parameter *x* is replaced immediately when the macro is called. The dummy parameter *z*, however, is not replaced until the IRP directive is processed. This means the dummy parameter is replaced as many times as there are numbers in the IRP parameter list. If the macro is called with

```
alloc var
```

the macro will be expanded as shown:

```
var1    DB    1
var2    DB    2
var3    DB    3
```

Literal Text Operator

The literal text operator (*< >*) directs MASM to treat a list as a single string rather than as separate arguments.

< text >

text is considered a single literal element even if it contains commas, spaces, or tabs. The literal text operator is most often used in macro calls and with the IRP directive to ensure that values in a parameter list are treated as a single parameter.

The literal text operator can also be used to force MASM to treat special characters, such as the semicolon or the ampersand, literally. For example, the semicolon inside angle brackets *<;>* becomes a semicolon, not a comment indicator.

MASM removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting. For example:

```
work    1,2,3,4,5    ; Passes five parameters
                    ;      to work
work    <1,2,3,4,5>  ; Passes one five element
                    ;      parameter to work
```

When the IRP directive is used inside a macro definition and when the argument list of the IRP directive is also a parameter of the macro, you must use the literal text operator (*< >*) to enclose the macro parameter.

For example, in the following macro definition, the parameter x is used as the argument list for the IRP directive:

```
init      MACRO      x
          IRP        y, <x>
          DB         y
          ENDM
          ENDM
```

If this macro is called with

```
init      <0,1,2,3,4,5,6,7,8,9>
```

the macro removes the angle brackets from the parameter so that it is expanded as 0,1,2,3,4,5,6,7,8,9. The brackets inside the repeat block are necessary to put the angle brackets back on. The repeat block is then expanded as shown below:

```
IRP      y, <0,1,2,3,4,5,6,7,8,9>
DB       y
ENDM
```

Literal Character Operator

The literal character operator (!) forces the assembler to treat a specified character literally rather than as a symbol. The syntax is:

!character

The literal character operator is used with special characters such as the semicolon or ampersand when meaning of the special character must be suppressed. Using the literal character operator is the same as enclosing a single character in brackets. For example, !! is the same as <!>

The following macro call is expanded to allocate the string Error 103: Expression > 255. Without the literal character operator, the greater than symbol would be interpreted as the end of the argument and an error would result.

```
errgen    MACRO      y,x
          PUBLIC     err&y
err&y     DB         'Error &y: &x'
          ENDM
          .
          .
          .
          errgen    103, <Expression !> 255>
```


Expression Operator

The expression operator `()` causes the assembler to treat the argument following the operator as an expression. The syntax is:

%text

MASM computes the expression's value and replaces *text* with the result. The expression can be either a numeric expression or a text equate. Handling text equates with this operator is a new feature in Version 5.0. Previous versions handled numeric expressions only. If there are additional arguments after an argument that uses the expression operator, the additional arguments must be preceded by a comma, not a space or tab.

The expression operator is typically used in macro calls when the programmer needs to pass the result of an expression rather than the actual expression to a macro. For example:

```
printe    MACRO    exp,val
           IF2      ;; On pass 2 only
           %OUT     exp = val    ;; Display expression and
           ENDIF    ;; result to screen
           ENDM

sym1      EQU      100
sym2      EQU      200
msg       EQU      <"Hello, World.">

           printe   <sym1 + sym2>,%(sym1 + sym2)
           printe   msg,%msg
```

In the first macro call, the text literal `sym1 sym2` is passed to the parameter `exp`, and the result of the expression is passed to the parameter `val`. In the second macro call, the equate name `msg` is passed to the parameter `exp`, and the text of the equate is passed to the parameter `val`. As a result, MASM displays the following messages:

```
sym1 + sym2 = 300
msg = Hello, World.
```

The `%OUT` directive, sends a message to the screen.

Macro Comments

A macro comment is any text in a macro definition that does not need to be copied in the macro expansion. A double semicolon (;;) is used to start a macro comment. The syntax is:

`;;text`

All *text* following the double semicolon (;;) is ignored by the assembler and will appear only in the macro definition when the source listing is created.

The regular comment operator (;) can also be used in macros. However, regular comments may appear in listings when the macro is expanded. Macro comments will appear in the macro definition, but not in macro expansions. Whether or not regular comments are listed in macro expansions depends on the use of the .LALL, .XALL, and .SALL directives, as described in Section 14.

Using Recursive, Nested, and Redefined Macros

The concept of replacing macro names with predefined macro text is simple, but in practice it has many implications and potentially unexpected side effects. The following subsections discuss advanced macro features and point out some side effects of macros:

- Using Recursion
- Nesting Macro Definitions
- Nesting Macro Calls
- Redefining Macros
- Avoiding Inadvertent Substitutions

Using Recursion

Macro definitions can be recursive: that is, they can call themselves. Using recursive macros is one way of doing repeated operations. The macro does a task, and then calls itself to do the task again. The recursion is repeated until a specified condition is met.

In the following example, the `pushall` macro repeatedly calls itself to push a register given in a parameter until no parameters are left to push. A variable number of parameters (up to six) can be given.

```
pushall    MACRO      reg1,reg2,reg3,reg4,reg5,reg6
            IFNB      <reg1>          ;; If parameter not blank
            push      reg1            ;; push one register and
                                      ;; repeat
            pushall    reg2,reg3,reg4,reg5,reg6
            ENDIF
            ENDM
            .
            .
            .
pushall    ax,bx,si,ds
pushall    cs,es
```

Nesting Macro Definitions

One macro can define another. MASM does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source file is assembled.

Using a macro to create similar macros can make maintenance easier. If you want to change all the macros, change the outer macro and it automatically changes the others.

The following macro, when called as shown, creates macros for multiple shifts with each of the shift and rotate instructions. All the macro names are identical except for the instruction.

```
shifts      MACRO      opname      ; Define macro that
defines macros
opname&s    MACRO      operand,rotates
            IF          rotates LE 4
            REPT        rotates
            opname      operand,1    ;; One at a time is faster
            ENDM        ;;          for 4 or less on
                        ;;          8088/8086
            ELSE
            mov          cl,rotates   ;; Using CL is faster
            opname      operand,cl    ;;          for more than 4 on
                        ;;          8088/8086
            ENDF
            ENDM
            ENDM

            shifts      ror           ; Call macro
            shifts      rol           ;          to new macros
            shifts      shr
            shifts      shl
            shifts      rcl
            shifts      rcr
            shifts      sal
            shifts      sar
            .
            .
            .
            shrs        ax,5          ; Call defined macros
            ror         bx,3
```

For example, the macro for the SHR instruction is called `shrs`; the macro for the ROL instruction is called `rols`. If you want to enhance the macros by doing more parameter checking, you can modify the original macro. Doing so will change the created macros automatically. This macro uses the substitute operator, as described in Substitute Operator.

Nesting Macro Calls

Macro definitions can contain calls to other macros. Nested macro calls are expanded like any other macro call, but only when the outer macro is called.

The sample macros `ex` and `express`, enable you to print the result of a complex expression to the screen by using the `OUT` directive, even though that directive expects text rather than an expression. Being able to see the value of an expression is convenient during debugging.

```
ex      MACRO      text,val      ; Inner macro definition
        IF2
        OUT        The expression (&text) has the value: &val
        ENDIF
        ENDM

express MACRO      expression ; Outer macro definition
ex      <expression> ,%(expression)
        ENDM
        .
        .
        .
        express    <4 + 2 * 7 - 3 MOD 4>
```

Both macros are necessary. The `express` macro calls the `ex` macro, using operators to pass the expression both as text and as the value of the expression. With the call in the example, the assembler sends the following line to the standard output:

The expression `(4 + 2 * 7 - 3 MOD 4)` has the value: 15

You could get the same output by using only the `ex` macro, but you would have to type the expression twice and supply the macro operators in the correct places yourself. The `express` macro does this for you automatically. Notice that expressions containing spaces must still be enclosed in angle brackets. Literal Text Operator explains why.

Redefining Macros

Macros can be redefined. You do not need to purge the macro before redefining it. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no statements or comments between the ENDM directive of the nested redefinition and the ENDM directive of the original macro.

The following macro allocates data space the first time it is called, and then redefines itself so that it doesn't try to reallocate the data on subsequent calls.

```
getasciiz    MACRO
              .DATA
char         DB ?
              .CODE
              push     ds
              push     OFFSET char
              call      ReadKbd
getasciiz    MACRO
              push     ds
              push     OFFSET char
              call      ReadKbd
              ENDM
            ENDM
```

Avoiding Inadvertent Substitutions

MASM replaces all parameters when they occur with the corresponding argument, even if the substitution is inappropriate. For example, if you use a register name such as AX or BH as a parameter, MASM replaces all occurrences of that name when it expands the macro. If the macro definition contains statements that use the register, not the parameter, the macro will be incorrectly expanded. MASM will not warn you about using reserved names as macro parameters.

MASM gives a warning if you use a reserved name as a macro name. You can ignore the warning, but be aware that the reserved name will no longer have its original meaning. For example, if you define a macro called ADD, the ADD instruction will no longer be available. Your ADD macro takes its place.

Managing Macros and Equates

Macros and equates are often kept in a separate file and read into the assembler source file at assembly time. In this way, libraries of related macros and equates can be used by many different source files as explained in the following subsections:

- Using Include Files
- Purging Macros from Memory

The **INCLUDE** directive is used to read an include file into a source file. Memory can be saved by using the **PURGE** directive to delete the unneeded macros from memory.

Using Include Files

The **INCLUDE** directive inserts source code from a specified file into the source file from which the directive is given. The syntax is:

INCLUDE *filespec*

filespec must specify an existing file containing valid assembler statements. When the assembler encounters an **INCLUDE** directive, it opens the specified source file and begins processing its statements. When all statements have been read, MASM continues with the statement immediately following the **INCLUDE** directive.

filespec can be given either as a file name, or as a complete or relative file specification including drive or directory name. If a complete or relative file specification is given, MASM looks for the include file only in the specified directory. If a file name is given without a directory or drive name, MASM looks for the file in the following order:

1. If paths are specified with the **/I** option, MASM looks for the include file in the specified directory or directories. See Section 4 for more information on the **/I** option.
2. MASM looks for the include file in the current directory.
3. If an **INCLUDE** environment variable is defined, MASM looks for the include file in the directory or directories specified in the environment variable.

Nested INCLUDE directives are allowed. MASM marks included statements with the letter “C” in assembly listings.

Note: *Any standard code can be placed in an include file. However, include files are usually used only for macros, equates, and standard segment definitions. Standard procedures are usually assembled into separate object files and linked with the main source modules.*

The syntax for the INCLUDE directive is as follows:

```
INCLUDE fileio.mac          ; File name only; use with  
                             ; /I or environment  
INCLUDE [d0] <sys>keybd.inc ; Complete file  
                             ; specification
```

Purging Macros from Memory

The PURGE directive can be used to delete a currently defined macro from memory. The syntax is:

```
PURGE macroname [ , macroname ] ..
```

Each *macroname* is deleted from memory when the directive is encountered at assembly time. Any subsequent call to that macro causes the assembler to generate an error.

The PURGE directive is intended to clear memory space no longer needed by a macro. If a macro has been used to redefine a reserved name, the reserved name is restored to its previous meaning.

The PURGE directive can be used to clear memory if a macro or group of macros is needed only for part of a source file.

It is not necessary to purge a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, a macro can purge itself as long as the PURGE directive is on the last line of the macro.

The PURGE directive works by redefining the macro to a null string. Therefore, calling a purged macro does not cause an error. The macro name is simply ignored.

These following example call a macro and then purge it. You might need to purge macros in this way if your system does not have enough memory to keep all the macros needed for a source file in memory at the same time.

```
GetStuff  
PURGE      GetStuff
```

Section 14

Controlling Assembly Output

The CTOS Microsoft Macro Assembler has two ways of communicating results of an assembly to the user. It can write information to a listing, cross reference, or object file; or it can display messages to the standard output device (ordinarily the screen). Both kinds of output can be controlled from the command line or from inside a source file. The command lines and options that affect information output are described in Section 4. This section explains the directives that directly control output from inside source files as follows:

- Sending Messages to the Standard Output Device
- Controlling Page Format in Listings
- Controlling the Contents of Listings
- Controlling Cross Reference Output

Sending Messages to the Standard Output Device

OUT directive instructs the assembler to display text to the standard output device. This device is normally the screen, but you can also redirect the output to a file or other device (see Section 4). The syntax is:

`%OUT text`

The *text* can be any line of ASCII characters. If you want to display multiple lines, you must use a separate OUT directive for each line.

The directive is useful for displaying messages at specific points of a long assembly. It can be used inside conditional assembly blocks to display messages when certain conditions are met.

The %OUT directive generates output for both assembly passes. The IF1 and IF2 directives can be used for control when the directive is processed. Macros that enable you to output the value of expressions are shown in Section 13.

```
IF1
%OUT          First Pass – OK
ENDIF
```

This sample block could be placed at the end of a source file so that the message First Pass – OK would be displayed at the end of the first pass, but ignored on the second pass.

Controlling Page Format in Listings

MASM provides several directives for controlling the page format of listings. These directives include the following:

Directive	Action
TITLE	Sets title for listings
SUBTTL	Sets title for sections in listings
PAGE	Sets page length and width, and controls page and section breaks

Setting the Listing Title

The TITLE directive specifies a title to be used on each page of assembly listings. The syntax is:

```
TITLE text
```

text can be any combination of characters up to 60 in length. The title is printed flush left on the second line of each page of the listing.

If no TITLE directive is given, the title will be blank. No more than one TITLE directive per module is allowed. The syntax is:

```
TITLE Graphics Routines
```

This above example sets the listing title. A page heading that reflects this title is shown below:

CTOS Microsoft (R) Macro Assembler Version 5.1.1

5/25/91 12:00:00

Graphics Routines

Page 1-2

Setting the Listing Subtitle

The SUBTTL directive specifies the subtitle used on each page of assembly listings. The syntax is:

SUBTTL *text*

The text can be any combination of characters up to 60 in length. The subtitle is printed flush left on the third line of the listing pages.

If no SUBTTL directive is used, or if no text is given for a SUBTTL directive, the subtitle line is left blank.

Any number of SUBTTL directives can be given in a program. Each new directive replaces the current subtitle with the new text. SUBTTL directives are often used just before a PAGE + statement, which creates a new section (see Section 14).

```
SUBTTL Point Plotting Procedure
PAGE      +
```

The example above creates a section title and then creates a page break and a new section. A page heading that reflects this title is shown below:

CTOS Microsoft (R) Macro Assembler Version 5.1.1

5/25/91 12:00:00

Graphics Routines

Page 3-1

Point Plotting Procedure

Controlling Page Breaks

The PAGE directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number accordingly, or to generate a page break in the listing.

```
PAGE [[length],width]
PAGE +
```

If *length* and *width* are specified, the PAGE directive sets the maximum number of lines per page to *length* and the maximum number of characters per line to *width*. The *length* must be in the range of 10 to 255 lines. The default page length is 50 lines. The *width* must be in the range of 60 to 132 characters. The default page width is 80 characters. To specify *width* without changing the default *length*, use a comma before *width*.

If no argument is given, PAGE starts a new page in the program listing by copying a form feed character to the file and generating new title and subtitle lines.

If a plus sign follows PAGE, a page break occurs, the section number is incremented, and the page number is reset to 1. Program listing page numbers have the following format:

section–*page*

The *section* is the section number within the module, and *page* is the page number within the section. By default, section and page numbers begin with 1–1. The SUBTTL directive and the PAGE directive can be used together to start a new section with a new subtitle.

The following will create a page break:

```
PAGE
```

The following sets the maximum page length to 58 lines and the maximum width to 90 characters.

```
PAGE 58,90
```

The following sets the maximum width to 132 characters. The current page length (either the default of 50 or a previously set value) remains unchanged.

```
PAGE ,132
```

The following creates a page break, increments the current section number, and sets the page number to 1. For example, if the preceding page was 3–6, the new page would be 4–1.

```
PAGE +
```

Controlling the Contents of Listings

MASM provides several directives for controlling what text will be shown in listings. The directives that control the contents of listings are shown below:

Directive	Action
<code>.LIST</code>	Lists statements in program listing
<code>.XLIST</code>	Suppresses listing of statements
<code>.LFCOND</code>	Lists false conditional blocks in program listing
<code>.SFCOND</code>	Suppresses false conditional listing
<code>.TFCOND</code>	Toggles false conditional listing
<code>.LALL</code>	Includes macro expansions in program listing
<code>.SALL</code>	Suppresses listing of macro expansions
<code>.XALL</code>	Excludes comments from macro listing

Suppressing and Restoring Listing Output

The `.LIST` and `.XLIST` directives specify which source lines are included in the program listing. The syntax is:

```
.LIST  
.XLIST
```

The `.XLIST` directive suppresses copying of subsequent source lines to the program listing. The `.LIST` directive restores copying. The directives are typically used in pairs to prevent a particular section of a source file from being copied to the program listing. The `.XLIST` directive overrides other listing directives such as `.SFCOND` or `.LALL`.

```
        .XLIST                                ; Listing suspended here  
        .  
        .  
        .  
        .LIST                                ; Listing resumes here  
        .  
        .  
        .
```

Controlling Listing of Conditional Blocks

The `.SFCOND`, `.LFCOND`, and `.TFCOND` directives control whether false conditional blocks should be included in assembly listings.

`.SFCOND`
`.LFCOND`
`.TFCOND`

The `.SFCOND` directive suppresses the listing of any subsequent conditional blocks whose condition is false. The `.LFCOND` directive restores the listing of these blocks. Like `.LIST` and `.XLIST`, conditional listing directives can be used to suppress listing of conditional blocks in sections of a program.

The `.TFCOND` directive toggles the current status of listing of conditional blocks. This directive can be used in conjunction with the `/X` option of the assembler. By default, conditional blocks are not listed on start up. However, they will be listed on start up if the `/X` option is given. This means that using `/X` reverses the meaning of the first `.TFCOND` directive in the source file. The `/X` option is discussed in Section 4.

In the following example, the listing status for the first two conditional blocks would be different, depending on whether the `/X` option was used. The blocks with `.SFCOND` and `.LFCOND` would not be affected by the `/X` option.

test1	EQU	0		
				; Defined to make all conditionals
				; false
				; /X not used /X used
	.TFCOND			
	IFNDEF	test1		; Listed Not listed
test2	DB	128		
	ENDIF			
	.TFCOND			
	IFNDEF	test1		; Not listed Listed
test3	DB	128		
	ENDIF			
	.SFCOND			
	IFNDEF	test1		; Not listed Not listed
test4	DB	128		
	ENDIF			
	.LFCOND			
	IFNDEF	test1		; Listed Listed
test5	DB	128		
	ENDIF			

Controlling Listing of Macros

The `.LALL`, `.XALL`, and `.SALL` directives control the listing of the expanded macros calls. The assembler always lists the full macro definition. The directives only affect expansion of macro calls.

The `.LALL` directive causes MASM to list all the source statements in a macro expansion, including normal comments (preceded by a single semicolon) but not macro comments (preceded by a double semicolon).

The `.XALL` directive causes MASM to list only those source statements in a macro expansion that generate code or data. For instance, comments, equates, and segment definitions are ignored.

The `.SALL` directive causes MASM to suppress listing of all macro expansions. The listing shows the macro call, but not the source lines generated by the call.

The `.XALL` directive is in effect when MASM first begins execution.

tryout	MACRO	param	
			;;Macro comment
			; Normal comment
it	EQU	3	; No code or data
	ASSUME	es:_DATA	; No code or data
	DW	param	; Generates data
	mov	ax,it	; Generates code
	ENDM		
	.		
	.		
	.		
	.XALL		
tryout		6	; Call with .LALL
	.XALL		
tryout		6	; Call with .XALL
	.SALL		
tryout		6	; Call with .SALL

The macro calls in the example generate the following listing lines:

```

                                .LALL
                                tryout      6      ; Call with
                                                ; .LALL
1                                ; Normal
                                ; comment
= 0003      1 it      EQU      3      ; No code or
                                ; data
                                1      ASSUME es:_TEXT ; No code or
                                                ; data
0015 0006      1      DW      6      ; Generates
                                ; data
0017 B8 0003      1      mov      ax,it ; Generates
                                ; code

                                .XALL
                                tryout      6      ; Call with
                                                ; .XALL
001A 0006      1      DW      6      ; Generates
                                ; data
001C B8 0003      1      mov      ax,it ; Generates
                                ; code

                                .SALL
                                tryout      6      ; Call with
                                                ; .SALL
```

Notice that the macro comment is never listed in macro expansions. Normal comments are listed only with the `.LALL` directive.

Controlling Cross Reference Output

The `.CREF` and `.XCREF` directives control the generation of cross references for the macro assembler's cross reference file. The syntax is:

`.CREF`

`.XCREF` [*name* [*name*]..]

The `.XCREF` directive suppresses the generation of label, variable, and symbol cross references. The `.CREF` directive restores generation of cross references. If names are specified with `.XCREF`, only the named labels, variables, or symbols will be suppressed. All other names will be cross referenced. The named labels, variables, or symbols will also be omitted from the symbol table of the program listing as shown in the following example.

```
.XCREF                ; Suppress cross referencing
.                    ; of symbols in this block
.
.
.CREF                ; Restore cross referencing
.                    ; of symbols in this block
.
.
.XCREF test1,test2    ; Don't cross reference test1 or test2
.                    ; in this block
.
.
```


Section 15

Understanding 8086 Family Processors

This section introduces the 8086 family of processors. It describes their segmented memory structure and their registers. Differences between the chips in the family are also covered. The following topics are covered:

- Using the 8086 Family Processors
- Segmented Addresses
- Using 8086 Family Registers

Using the 8086 Family Processors

The Intel Corporation manufactures the group of processors referred to in this manual as the 8086 family processors. These are discussed in the following two subsections:

- Processor Differences
- Real and Protected Modes

The processors have several features in common, as follows:

- Memory is organized by using a segmented architecture.
- The instruction set is upwardly compatible. All features available in the early versions of the processor are also available in the newer versions, but the new versions contain additional features not supported in the old versions.
- The register set is also upwardly compatible.

Processor Differences

The main 8086 family processors are discussed below:

Processor	Description XP
-----------	----------------

8088 and 8086	These processors work in real mode. They are designed to run a single process. No provision is made to protect one part of memory from actions occurring in another part of memory. The processor can address up to one megabyte of memory. Addresses specified in assembly language correspond to physical memory addresses.
---------------	---

The 8088 uses an 8 bit data bus, and the 8086 uses a 16 bit data bus. This makes the 8086 somewhat faster. However, from the programming standpoint, the two processors are identical except that the 8086 will handle certain data more efficiently if you word align it by using the EVEN or ALIGN directives (See Section 8).

80186	This processor is identical to the 8086 except that new instructions have been added and some old instructions have been optimized. It runs significantly faster than the 8086. (There is also an enhanced version of the 8088 called the 80188.)
-------	---

80286	This processor has the added instructions and speed of the 80186. It can run in the real mode of the 8088 and 8086, but it also has an optional protected mode in which multiple processes can be run concurrently. Memory used by each process can be protected from other processes.
-------	--

In protected mode, the processor can address up to 16 megabytes of memory. However, when memory is accessed in protected mode, the addresses do not correspond to physical memory. Under protected mode operating systems, the processor allocates and manages memory dynamically. Additional privileged instructions for initializing protected mode and controlling multiple processes are available.

- 80386 This is both a 16 bit and a 32 bit processor. It is fully compatible with the 80286; but at the system level, it implements many new features, including virtual memory, multiple 8086 processes, and addressing for up to four gigabytes of memory. This manual does not explain how to use these features.
- The 80386 supports all the instructions of the 80286 and some additional instructions. It also allows limited use of 32 bit registers and addressing modes. Finally, the 80386 operates significantly faster than the 80286.
- 8087,
80287, and These are math coprocessors that work concurrently with
80387 the 8086 family processors. They do mathematical calculations faster and more accurately than can be done with the 8086 family processors. Although there are performance and technical differences between the three coprocessors, the main difference to the applications programmer is that the 80287 and 80387 can operate in protected mode. The 80387 also has several new instructions.

Real and Protected Modes

To the applications programmer, there is little difference between assembly language programming in real or protected mode. Processes are managed at the system level by the operating system. The applications programmer does not deal with processes except when interfacing with the operating system.

This manual does not address issues of interfacing with multitasking operating systems. If you are using a multitasking system, you must use the documentation for that operating system. However, applications programmers should be aware of the following differences between real and protected mode programming:

- In protected mode, up to 16 megabytes of memory can be addressed (compared to one megabyte in real mode). This distinction may make a difference in the number and size of data structures created, but it should make no difference in the assembly language syntax, since data is addressed in exactly the same way in either mode.
- In protected mode, segment registers contain segment selectors rather than actual segment values. The selectors must come from the operating system. They cannot be calculated by the program. Programming techniques that attempt to calculate segment values or address memory directly will not work.
- Certain instructions that can be used normally in real mode are privileged instructions in protected mode operating systems. These include STI, CLI, IN, and OUT. These instructions are still available at privilege levels normally used only by systems programmers.

Protected mode operating systems, provide extended functions for doing the kinds of tasks that are currently done by using the restricted practices described above.

Segmented Addresses

8086 family processors can store addresses as 16 bit word values. Therefore, the maximum unsigned value that can be stored as an address is 65,535 (0FFFFh). Yet the processors are actually capable of accessing much larger addresses. The highest possible address is one megabyte (0FFFFFFh) in real mode or 16 megabytes (0FFFFFFFh) in protected mode.

Addresses larger than 65,535 bytes are specified by combining two segmented word addresses: a 16 bit segment and a 16 bit offset within the segment. A common syntax for showing segmented addresses is the segment:offset format. For example, an address with a segment of 053C2h and an offset of 0107Ah would be represented as 53C2:107A. This method of specifying addresses can be used directly in most debuggers, but it is not legal in assembler source code.

In real mode, the address 53C2:107A represents a physical 20 bit address. This address can be calculated by multiplying the segment portion of the address by 16 (10h), and then adding the offset portion, as shown:

53C20h	Segment times 10h
+ 107Ah	Offset
<hr/> 54C9Ah	Physical address

In protected mode, the address 53C2:107A represents a movable address. The segment portion of the address is a selector assigned a physical address by the operating system. The applications programmer has no control (and needs none) over the physical address represented by the selector.

Note: *The 80386 processor supports 48 bit addresses consisting of a 16 bit segment selector and a 32 bit offset. This enables the processor to access addresses of up to four gigabytes per segment in protected mode. The processor can also run in modes compatible with the 16 bit real and protected mode addressing schemes of the other 8086 family processors.*

Addresses cannot be represented directly in the segment:offset format in assembly language. Instead the segment portion of the address is specified symbolically, using a name assigned to the segment in the source code. The address represented by the symbol can then be assigned to one of the segment registers. Section 7 describes the directives that assign symbols to segment addresses.

The offset portion of addresses can be specified in a number of ways, depending on the context. Directives that assign symbols to offsets are discussed in section 8.

In assembly language programming, addresses can be near or far. A near address is simply the offset portion of the address. Any instruction that accesses a near address will assume that the segment address is the same as the current segment for the type of address being accessed (usually a code segment for code or a data segment for data).

A far address consists of both the segment and offset portions of the address. Far addresses can be accessed from any segment. Both the segment and offset must be provided for instructions that access far addresses. Far addresses are more flexible because they can be used for larger programs and larger data objects. However, near addresses are more efficient, since they produce smaller code and can be accessed more quickly.

Using 8086 Family Registers

Like most microprocessors, the 8086 family processors have special areas of memory called registers. Some registers control the behavior or status of the processor. Others are used as temporary storage places where data can be accessed and processed faster than if data were stored in regular memory. Registers are discussed in the following subsections:

- Segment Registers
- General Purpose Registers
- Other Registers
- The Flags Register
- 8087 Family Registers

All the 8086 family processors share the same set of 16 bit registers. Some registers can be accessed as two separate 8 bit registers. In the 80386, most registers can also be accessed as extended 32 bit registers.

Figure 15–1 shows the registers common to all the 8086 family processors. Each register and group of registers has its own special uses and limitations, as described in this section.

Note: *The 80386 processor uses the same registers as the other processors in the 8086 family, but all except the segment registers can be extended to 32 bits. The extended registers begin with the letter E. For example, the 32 bit version of AX is EAX. The 80386 also has two additional segment registers, FS and GS. Figure 15–2 shows the extended registers of the 80386.*

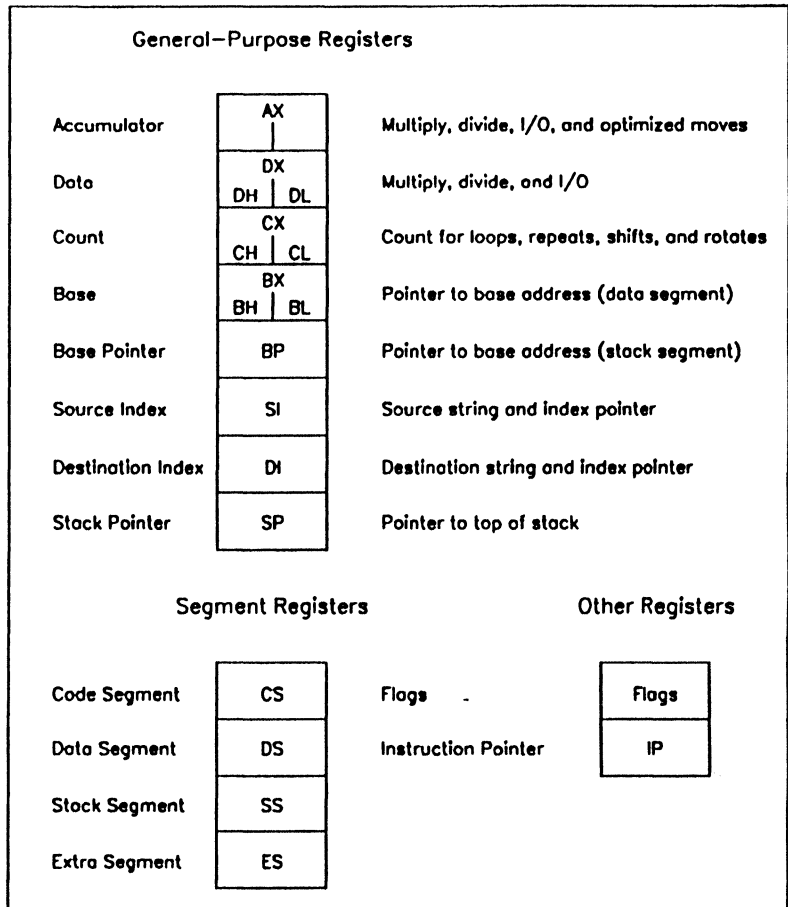


Figure 15-1. Register for 8088-80286 Processors

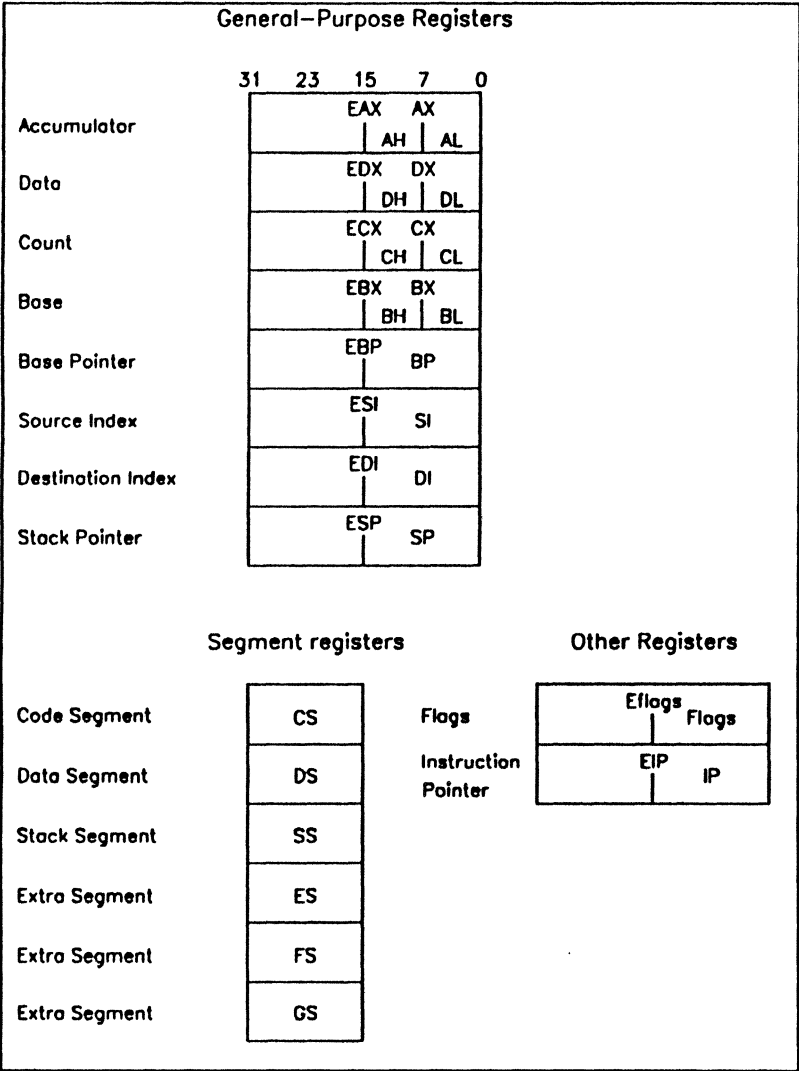


Figure 15-2. Extended Registers of 80386 Processor

Segment Registers

At run time, all addresses are relative to one of four segment registers: CS, DS, SS, or ES. These registers and the segments they correspond to are listed in Table 15–1.

Table 15–1. Segment Registerst

Segment	Purpose
Code Segment (CS)	Addresses in the segment pointed to by this register contain the encoded instructions and operands specified by the program.
Data Segment (DS)	Addresses in the segment pointed to by this register normally contain data allocated by the program.
Stack Segment (SS)	Addresses in the segment pointed to by this register are available for instructions that store data on the program stack. A stack is an area of memory reserved for storing temporary data. See Transferring Data to and from the Stack, for information on using stacks.
Extra Segment (ES)	Addresses in the segment pointed to by this register are available for string instructions. ES can also be used to address a secondary data segment. The 80386 has two additional extra segments, FS and GS.

General Purpose Registers

The AX, DX, CX, BX, BP, SI, and DI registers are 16 bit, general purpose registers. They can be used to temporarily store data during processing. Data in registers can be accessed much more quickly than data in memory. Therefore, it is more efficient to keep the most frequently used values in registers.

Memory to memory operations are never allowed in 8086 family processors. As a result, data must often be moved into registers before doing calculations or other operations involving more than one variable.

Four of the general registers, AX, DX, CX, and BX, can be accessed as two 8 bit registers or as a single 16 bit register. The AH, DH, CH, and BH registers represent the high order 8 bits of the corresponding registers. Similarly, AL, DL, CL, and BL represent the low order 8 bits of the registers. All the general registers can be extended to 32 bits on the 80386 by appending the letter E for example, EAX, EDX, ECX, and so on.

In addition to their general use for storing data, each of the general purpose registers has special uses in certain situations. Specific uses for each register are listed in Table 15–2.

Table 15–2. General Purpose Register

Register	Description XP
AX	<p>The AX (Accumulator) register is most often used for storing temporary data. Many instructions are optimized so that they work slightly faster on data in the accumulator register than on data in other registers.</p> <p>With division instructions, the accumulator holds all or part of the dividend before the operation and the quotient afterward. With multiplication instructions, the accumulator holds one of the factors before the operation and all or part of the result afterward. In I/O operations to and from ports, the accumulator holds the data being transferred.</p>
DX	<p>The DX (Data) register is most often used for storing temporary data.</p> <p>When dividing a doubleword value, DX holds the upper word of the dividend before the operation and the remainder afterward. When multiplying word values, DX holds the upper word of the doubleword result. In I/O operations to and from ports, DX holds the number of the port to be accessed.</p>
CX	<p>The CX (Count) register must be used to hold the count for instructions that do looping or other repeated operations. These include the loop instructions, certain jump instructions, repeated string instructions, and shifts and rotates. This register can also be used for temporary data storage.</p>
BX	<p>The BX (Base) register can be used as a pointer. For instance, it can point to the base of a data object (see Indirect Memory Operands). This register can also be used for temporary data storage.</p>

continued

Table 15–2. General Purpose Register (cont.)

Register	Description XP
BP	The BP (Base Pointer) register can be used for general data storage. It is more often used as a pointer. For instance, it is often used to point to the base of a stack frame. The Microsoft conventions for passing arguments to procedures have a specific use for BP as described in Passing Arguments on the Stack. The SS register is assumed as the segment register in operations using BP.
SI	The SI (Source Index) register can be used as a pointer or for general data storage. It is often used for pointing to (indexing) an item within a data object. With string instructions, SI is used to point to bytes or words within a source string.
DI	The DI (Destination Index) register can be used as a pointer or for general data storage. It is often used for pointing to (indexing) an item within a data object. With string instructions, DI is used to point to bytes or words within a destination string.

Other Registers

The 8086 family processors have two additional registers whose values are changed automatically by the processor.

Register	Description
----------	-------------

SP	<p>The SP (Stack Pointer) register points to the current location within the stack segment. Pushing a value onto the stack decreases the value of SP by two; popping from the stack increases the value of SP by two. Call instructions store the calling address on the stack and decrease SP accordingly; return instructions get the stored address and increase SP. With 80386 32 bit segments, SP is increased or decreased by four instead of two. Using the Stack, and Passing Arguments on the Stack, discuss operation of the stack in more detail.</p>
----	--

SP is technically a general purpose register that could be used in calculations or for temporary data storage. However, it should generally be used only for stack operations.

IP The IP (Instruction Pointer) register always contains the address of the instruction about to be executed. The programmer cannot directly access or change the instruction pointer. However, instructions that control program flow (such as calls, jumps, loops, and interrupts) automatically change the instruction pointer.

The Flags Register

The flags register is a 16 bit register made up of bits that control various instructions and reflect the current status of the processor. In the 80386 processor, the flags register is extended to 32 bits. Some bits are undefined, so there are actually 9 flags for real mode, 11 flags (including a 2 bit flag) for 80286 protected mode, and 13 flags for the 80386. The extended flags register of the 80386 is sometimes called eflags.

Figure 15–3 shows the bits of the 32 bit flags register for the 80386. Only the lower word is used for the other 8086 family processors. The unmarked bits are reserved for processor use and should never be modified by the programmer.

The nine flags common to all 8086 family processors are summarized below, starting with the low order flags. In these descriptions, the term set means the bit value is 1, and cleared means the bit value is 0.

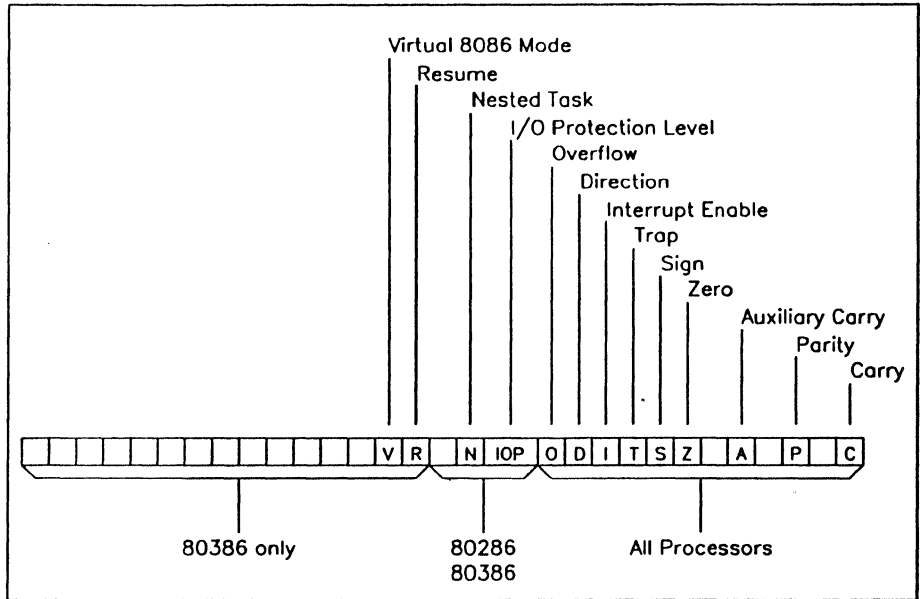


Figure 15-3. Flags for 8088-80386 Processors

Table 15-3. Flags

Flag	Description
Carry	Is set if an operation generates a carry to or a borrow from a destination operand.
Parity	Is set if the low order bits of the result of an operation contain an even number of set bits.
Auxiliary Carry	Is set if an operation generates a carry to or a borrow from the low order four bits of an operand. This flag is used for binary coded decimal arithmetic.
Zero	Is set if the result of an operation is 0.
Sign	Equal to the high order bit of the result of an operation (0 is positive, 1 is negative).

continued

Table 15–3. Flags (cont.)

Flag	Description
Trap	If set, the processor generates a single step interrupt after each instruction. A debugger program can use this feature to execute a program one instruction at a time.
Interrupt Enable	If set, interrupts will be recognized and acted on as they are received. The bit can be cleared to temporarily turn off interrupt processing.
Direction	Can be set to make string operations process down from high addresses to low addresses, or can be cleared to make string operations process up from low addresses to high addresses.
Overflow	Is set if the result of an operation is too large or small to fit in the destination operand.
I/O Protection Level	This 2 bit flag indicates the protection level for input and output. Managing the protection level is a systems task not described in this manual.
Nested Task	Controls chaining of interrupted and called tasks. Controlling tasks in protected mode is a systems task not described in this manual.
Resume	If set, debug exceptions are temporarily disabled. Using 80386 debug exceptions is a systems task not described in this manual.
Virtual 8086 Mode	If set, the processor is running an 8086 family real mode program in a protected multitasking environment. If clear, the 80386 processor is in its normal mode. Running in virtual 8086 mode is a systems task not described in this manual.

8087 Family Registers

The 8087 family processors use a stack based architecture to access up to eight 80 bit registers. See Section 21 for information on using 8087 family registers and instructions. The format of real numbers used by coprocessors is explained in Section 8.

Using the 80386 Processor

Applications programmers can use some 80386 enhancements. Note that using any of these features means your code will not run on machines that do not have an 80386 processor.

- You can use the new 80386 instructions (except for those that manage protected mode). New instructions include bit scan (BSF and BFR); bit test (BT, BTC, BTR, and BTS); move with sign and zero extend (MOVSX and MOVZX); set byte on condition (SET condition); and double precision shift (SHLD and SHRD).
- You can use 80286 instructions that have been enhanced to work with 32 bit registers. These include the integer multiply instruction (IMUL); conversion instructions (CWDE and CDQ); string instructions (CMPSD, LODSD, MOVSD, SCASD, STOSD, INSD, OUTSD); and 32 bit stack enhancements (PUSHAD, POPAD, PUSHFD, POPFD, and IRETD).
- You can use 32 bit registers for calculations. For instance, you can add and subtract doubleword integers without using multiple registers, and you can do some multiplication and division operations on 64 bit integers.
- You can use 32 bit registers to point into 16 bit segments. In previous processors, only BX, BP, DI, and SI could be used as pointers in indirect memory operands. The 80386 has the same limitations on 16 bit registers, but allows any general purpose 32 bit register to be a pointer in an indirect memory operand. If you use this technique, you must make sure that 32 bit registers used as pointers actually contain valid 16 bit addresses.
- If a program that uses 32 bit registers needs to execute while another program is running, it should save 32 bit registers at entry and restore them when finished. This applies to programs that interrupt other programs. Use PUSHAD to save and POPAD to restore. Without these instructions, the interrupting program may change the upper half of 32 bit registers, causing errors when the interrupted program regains control.

Section 16

Using Addressing Modes

Instruction operands can be given in different forms called addressing modes. Addressing modes tell the processor how to calculate the actual value of an operand at run time. The following subsections describe the three kinds of addressing modes:

- Immediate
- Register
- Memory

Memory operands are further broken into two groups, direct and indirect memory operands.

The value of operands is calculated at assembly time for immediate operands, at load time for direct memory operands, and at run time for register operands and indirect memory operands.

Although two statements may be similar and their instruction mnemonic the same, MASM may actually assemble different code for an instruction when it is used with different addressing modes. For example, the statements:

	<code>mov</code>	<code>ax,1</code>
and	<code>mov</code>	<code>ax,place[bx][di]</code>

use the same instruction, but have different encoding, timing, and size.

Instructions that take two or more operands always work right to left. The right operand is the source operand. It specifies data that will be used, but not changed, in the operation. The left operand is the destination operand. It specifies the data that will be operated on and possibly changed by the instruction.

Using Immediate Operands

Immediate operands consist of constant numeric data that are known or calculated at assembly time. Immediate values are coded into the executable program and processed the same way each time the program is run.

Some instructions have limits on the size of immediate values (usually 8, 16, or 32 bit). String constants longer than two characters (four characters on the 80386) cannot be immediate data. They must be stored in memory before they can be processed by instructions.

Many instructions permit immediate data in the source (right) operand and either memory or register data in the destination (left) operand. The instruction combines or replaces the register or memory data with the immediate data in some way defined by the instruction. Examples of this type of instruction include MOV, ADD, CMP, and XOR.

A few instructions, such as RET and INT, take a single immediate operand.

Immediate data is never permitted in the destination operand. If the source operand is immediate, the destination operand must be either register or direct memory so that there will be a place to store the result of the operation. For example:

```
.DATA
five      DB          5          ; Memory data
nine      EQU         9          ; Constant data

.CODE
.
.
.
; Source operand is immediate
mov        bx,nine+3
add        five,3
or         bx,00100100b
in         al,43h
cmp        cx,200

; Only operand is immediate
ret        6
int        100h
```

Using Register Operands

Register operands consist of data stored in registers. Register direct mode refers to using the actual value inside the register at the time the instruction is used. Registers can also be used indirectly to point to memory locations, as described in Indirect Memory Operands.

Most instructions allow register values in one or more operands. Some instructions can only be used with certain registers. Often instructions have shorter encoding (and faster operation) if the accumulator register (AX or AL) is specified. Use of segment registers in operands is limited to a few instructions and special circumstances.

The registers shown in Table 16–1 can be used in register direct mode.

Table 16–1. Register Operands

Registered Operand Type	Register	Name			
8 bit high register	AH	BH	CH	DH	
8 bit low registers	AL	BL	CL	DL	
16 bit general purpose	AX	BX	CX	DX	
32 bit general, pointer, and index ¹	EAX	EBX	ECX	EDX	
16 bit pointer and index	SP	BP	SI	DI	
32 bit general, pointer, and index ¹	ESP	EBP	ESI	EDI	
16 bit segment	CS	DS	SS	ES	
Additional 80386 segment ¹	FS	GS			

¹Available only if the 80386 processor is enabled

Registers are discussed in more detail in Section 15. Limitations on register use for specific instructions are discussed in sections on the specific instructions throughout Using Instructions.

```
; Source and destination operands are register direct
    add     ax,bx
    mov     ds,ax
    xor     eax,ebx           ; 80386 only
    cmp     ah,bh

; Source operand is register direct
    and     stuff,dx
    sub     array[bx][si],ax

; Destination operand is register direct
    shl     ax,1
    cmp     cx,counter

; Only operand is register direct
    mul     bx
    pop     cx
    inc     ah
```

Using Memory Operands

Many instructions can work on data in memory. When a memory operand is given, the processor must calculate the address of the data to be processed. This address is called the effective address. Calculation of the effective address depends on how the operand is specified, as explained in the following subsections:

- Direct Memory Operands
- Indirect Memory Operands
- 80386 Indirect Memory Operands

Memory to memory operations are never allowed. These operations must be done indirectly by moving one of the memory values into a register before processing it.

Direct Memory Operands

A direct memory operand is a symbol that represents the address (segment and offset) of an instruction or data. The offset address represented by a direct memory operand is calculated at assembly time. The address of each operand relative to the start of the program is calculated at link time. The actual (or effective) address is calculated at load time.

Direct memory operands can be any constant or symbol representing an address. This includes labels, procedure names, variables, structure variables, record variables, or the value of the location counter.

The effective address is always relative to a segment register. The default segment register is DS for direct memory operands, but the default segment can be overridden with the segment override operator (:), as explained in Section 11.

Direct memory operands are often specified as constant expressions by using the index operator. For example, the operand `table[4]` refers to the byte having an offset four bytes from the address of `table`. This expression is equivalent to `table+4`.

The following example illustrates the difference between memory operands that represent addresses and memory operands that represent the value at an address. Labels and variable names in the data segment (such as `stuff`) represent the value at an address. Code labels (such as `here`) represent the address itself. The four jump statements at the end of the example use different kinds of operands to transfer control to the same address.

```
stuff      .DATA
           DW here
           .CODE
           .
           .
           mov     ax,stuff      ; Load value at address stuff
                                   ; (address of here) into AX
           mov     bx,OFFSET stuff ; Load address of stuff
                                   ; into BX
           jmp     'stuff        ; Jump to value of stuff
                                   ; (which is address of
                                   ; here)
           jmp     here          ; Jump to the address of here
           jmp     ax            ; Jump to AX (value of stuff)
           jmp     [bx]          ; Jump to [BX] (value at
                                   ; address
                                   ; of stuff)
           .
           .
here:
```


If the label is omitted from a direct memory operand used with a constant index, a segment must be specified. The offset of the operand is assumed to be the start of the specified segment plus the indexed offset. For example:

```
mov     ax,ds:[100h]
```

moves the value at address 100h in the data segment into the AX register. It is equivalent to

```
mov     ax,ds:100h
```

If the segment override is omitted, the constant (immediate) value of the operand is used rather than the value it points to. For example,

```
mov     ax,[100h]
```

moves the value 100h into the AX register. It is equivalent to the statement

```
mov     ax,100h
```

Indirect Memory Operands

Indirect memory operands enable you to use registers to point to values in memory. Since values in the registers can change at run time, you can use indirect memory operands to operate on data dynamically.

On all processors except the 80386, only four registers can be used in indirect mode (see 80386 Indirect Memory Operands for information on 80386 enhancements). BX and BP are called base registers; DI and SI are called index registers. The distinction between base and index registers is not always important. In many contexts, any of these registers can be thought of as the base or the index. In any case, an attempt to use any register other than these four in a statement that accesses memory indirectly results in an error.

You can use the base and index registers separately or in pairs, with or without specifying a displacement. A displacement can be either a constant or a direct memory. Several displacements can be given, but they are all added into a single displacement at assembly time. For example, in the statement

```
mov     ax,table[bx][di]+6
```

both table and 6 are displacements. MASM calculates the actual offset of table plus 6 to get the total displacement.

The modes in which registers can be used to specify indirect memory operands are shown in Table 16–2.

Table 16–2. Indirect Addressing Modes

Mode	Syntax	Description
Register indirect	[BX] [BP] [DI]	Effective address is contents of register
Based or indexed	displacement[BX] displacement[BP] displacement[DI] displacement[SI]	Effective address is contents of register and displacement
Based indexed	[BX][DI] [BP][DI] [BX][SI] [BP][SI]	Effective address is contents of base register and contents of index register
Based indexed with displacement	displacement[BX][DI] displacement[BP][DI] displacement[BP][SI]	Effective address is contents of base register and contents of index registers and displacement

Register indirect operands are typically used to point to a memory address within a segment. Based and indexed operands are used to point to a memory address relative to a table, a one dimensional array, or a structure. Operands with multiple indexes are useful for pointing to memory locations in complex data structures such as multidimensional arrays.

The choice of which registers to use depends on the context of the statement. String instructions require that specific registers are used in specific situations, as explained in Processing Strings. With other instructions, base and index registers can often be used interchangeably, depending on which registers are available.

When calculating the effective address of an indirect operand, the processor uses DS as the default segment register if BX is used as a base register, or if no base register is specified. If BP is used anywhere in the operand, the default segment register is SS. The default segment can be overridden with the segment override operator (:), as explained in Section 11.

A common syntax for indirect memory operands is each register put within index operators ([]). The register or registers must always be within brackets, but a variety of alternate syntaxes is possible. Any operator that indicates addition can be used to combine the displacement and multiple registers. For example, the following statements are equivalent:

```
mov    ax,table[bx][di]
mov    ax,table[bx+di]
mov    ax,[table+bx+di]
mov    ax,[bx][di].table
mov    ax,[bx][di]+table
mov    ax,table[di][bx]
```

When using based indexed modes, one of the registers must be a base register and the other an index register. The following statements are illegal:

```
mov    ax,table[bx][bp]      ; Illegal – two base
                               ; registers
mov    ax,table[di][si]      ; Illegal – two index
                               ; registers
```

Use of the index operator is explained in more detail in Section 11.

When an index or displacement points into an array, it must be scaled for the size of elements in the array. On all processors except the 80386, scaling must be done in separate statements (see 80386 Indirect Memory Operands, for information on 80386 scaling). The scaling factor is 1 for bytes (no scaling necessary), 2 for words, 4 for doublewords, and 8 for quadwords. Since scaling factors (other than for bytes) are multiples of 2, they can usually be calculated quickly with the SHL instruction, as shown below:

```
shl    di,1                  ; Scale DI for words (DI*2)
shl    di,1                  ; Scale DI for doublewords (DI*4)
shl    di,1
shl    di,1
shl    di,1                  ; Scale DI for quadwords (DI*8)
shl    di,1
shl    di,1
```

Use of the SHL instruction for multiplication is described in more detail in Multiplying and Dividing by Constants.

add	dx, [bx]	; Add the word contents of ; DS:BX ; to the contents of DX
mov	dl, [bp+6]	; Load the byte contents ; of SS:BP+6 into DL
sub	dx, 12[bx]	; Subtract the word ; contents of ; DS:12+BX from the ; contents of DX
xor	red[bx], dx	; XOR the contents of DX ; with ; the contents of ; DS:red+BX
and	dx, red[si] + 3	; AND the contents of ; DS:red+SI + 3 ; with the contents of ; DX
dec	BYTE PTR [bx][si]	; Decrement the byte ; at DS:BX+SI
cmp	cx, here[bp][si]	; Compare the contents of ; CX ; to the contents of ; SS:here+BP+SI
push	place[bx][di] + 2	; Save the contents of ; DS:place+BX+DI+2 on ; the stack
call	cs:table[bx]	; Call the routine pointed ; to ; by the contents of ; CS:table+bx

The statements in the above example illustrate how the various instructions can be used with indirect memory operands.

Using Addressing Modes

```
scrnbuf  EQU      0B800h           ; CGA screen buffer (actual
                                     ; value is hardware dependent)
mov      ax,scrnbuf                ; Load address of screen
mov      es,ax                     ;   buffer
                                     ;   into ES
mov      ax,4                       ; Push column 4 as third
                                     ; argument
push     ax
mov      ax,6                       ; Push row 6 as second
                                     ; argument
push     ax
mov      ax,"z"                     ; Push z as first argument
push     ax
call     show                       ; Call the procedure
add      sp,6                       ; Restore stack
.
.
.
show     PROC      NEAR
push     bp                         ; Save BP
mov      bp,sp                     ; and set up stack frame
push     si                         ; Save SI (so procedure
                                     ; could
                                     ; be called from C)

mov      si,[bp+8]                  ; Load column
dec      si                         ; Adjust for zero
shl      si,1                       ; Scale for 2 bytes per
                                     ; character
mov      bx,[bp+6]                  ; Load row
dec      bx                         ; Adjust for zero
mov      ax,160                     ; Multiply 160 bytes per
                                     ; line
mul      bx                         ; times current row
mov      bx,ax                     ; Put result in index

mov      dl,BYTE PTR [bp+4]         ; Load character
mov      es:[bx][si],dl             ; Put character in buffer

pop      si                         ; Restore SI and BP
pop      bp
ret                                     ; Return
show     ENDP
```

The above example illustrates two uses of indirect memory operands. Arguments are pushed onto the stack before calling a procedure. When the procedure is called, the arguments are removed using indirect memory operands.

The procedure writes a character to a screen buffer (a common technique with many computers and display adapters). The BX register points to the column position in the buffer; the SI register points to the row position. In this example, the ES register must contain the address of the screen buffer (this address varies for different hardware).

The procedure follows the calling conventions of Microsoft C and could be called directly from that language. Note that SI is saved and restored because the C compiler requires that it not be changed by a procedure.

The second example works on any processor. 80386 Indirect Memory Operands, shows an enhanced version that uses 80386 instructions and addressing modes.

80386 Indirect Memory Operands

Instructions for the 80386 can be given in two modes, 16 bit and 32 bit. Understanding these modes is important, since indirect memory operands are different in each mode.

The 80386 instruction modes are controlled by the use type of the code segment in which the instructions are located. The mode is 16 bit if the use type is USE16 or 32 bit if the use type is USE32. In 32 bit mode, an offset address can be up to four gigabytes. In 16 bit mode, an offset address can be up to 64K. The 16 bit mode of the 80386 is the same as the mode used by all the other 8086 family processors.

If the 80386 processor is enabled (with the .386 directive), 32 bit general purpose registers are always available. They can be used from 16 bit or 32 bit segments. When 32 bit registers are used, many of the limitations of 16 bit indirect memory modes do not apply. The following extensions are available when 32 bit registers are used in indirect memory operands:

- There are fewer limitations on the registers that can be used as base and index registers. With other 8086 family processors, only BX, BP, DI, and SI registers can be used in indirect memory operands. With the 80386, any general purpose 32 bit register can be used. The same register can even be used as both the base and the index. Several examples are shown below:

add	edx,[eax]	; Add double
mov	dl,[esp+10]	; Add byte from stack
dec	WORD PTR [edx][eax]	; Decrement word
cmp	cx,array[edx][eax]	; Compare word from array
jmp	table[ecx]	; Jump into pointer table

- The index register can have a scaling factor of 1, 2, 4, or 8. Any register except ESP can be the index register and can have a scaling factor. The scaling factor is specified by using the multiplication operator (*) adjacent to the register.

Scaling can be used to index into arrays with different sizes of elements. For example, the scaling factor is 1 for byte arrays (no scaling needed), 2 for word arrays, 4 for doubleword arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor. Scaling is illustrated in the following examples:

```
mov     eax,darray[edx*4]           ; Load double of double array
mov     eax,[esi*8][edi]            ; Load double of quad array
mov     ax,wtbl[ecx+2][edx*2]       ; Load word of word array
```

- The default segment register is SS if the base register is EBP or ESP; it is DS for all other the base registers. If two registers are used, only one can have a scaling factor and it is defined to be the index register. The other register is the base. If scaling is not used, the first register is the base. If one register is used, it is the base, regardless of scaling. The following examples illustrate how to determine the base register:

```
mov     eax,[edx][ebp*4]            ; EDX base (not scaled) –
                                     ; DSsegment
mov     eax,[edx*1][ebp]            ; EBP base (not scaled) – SS
                                     ; segment
mov     eax,[edx][ebp]              ; EDX base (first) – DS segment
mov     eax,[ebp][edx]              ; EBP base (first) – SS segment
mov     eax,[ebp*2]                 ; EBP base (only) – SS segment
```

Statements can mix 16 and 32 bit registers. However, it is important to understand the implications of these statements. For example, the following statement is legal for either 16 or 32 bit segments:

```
mov     eax,[bx]
```

This moves the 32 bit value pointed to by BX into the EAX register. Although BX is a 16 bit pointer, it may still point into a 32 bit segment. However, the following statement is never legal:

```
mov     eax,[cx]
```

The CX register may not be used as a 16 bit pointer (although ECX may be used as a 32 bit pointer). The following statement is also legal in either mode:

```
mov     bx,[eax]
```

This moves the 16 bit value pointed to by EAX into the BX register. This works fine in 32 bit mode; but in 16 bit mode, a 32 bit pointer moved into a 16 bit segment may cause problems. If EAX contains a 16 bit value (the top half of the 32 bit register is 0), then the statement works. However, if the top half of the EAX register is not 0, the processor may generate an error.

WARNING

It is possible to use both 16 bit and 32 bit modes in the same program by defining separate code segments for the two modes. However, this is a complex technique that involves special calculations to account for the differences between the two modes. Combining modes is generally done only in systems programming and is beyond the scope of this manual.

This example is the same as the one in Indirect Memory Operands, except that it uses enhanced 80386 instructions and addressing modes to make the code shorter and more efficient.

```

.MODEL      small                ; .MODEL preceeds .386
.386                ; to make 16 bit
                        ; segments

scrnbuff EQU      0B800h        ; CGA screen buffer (actual
                                ; value is hardware
                                ; dependent)

.CODE
.
.
.
    mov     ax,scrnbuff        ; Load address of screen buffer
    mov     es,ax              ; into ES
    push    4                  ; Push column 4 as third
                                ; argument
    push    6                  ; Push line 6 as second
                                ; argument
    push    "z"                ; Push z as first
                                ; argument
    call    show               ; Call the procedure
    add     sp,6               ; Restore stack
.
.
.
show PROC          NEAR

```



```
movzx    ebx,WORD PTR [esp+6]    ; Load column
dec      ebx                    ; Adjust for zero
movzx    eax,WORD PTR [esp+4]    ; Load row
dec      eax                    ; Adjust for zero
imul     eax,160                 ; Multiply 160 bytes per
                                   ; line

mov      dl,[esp+2]              ; Load character
mov      es:[eax][ebx*2],dl      ; Put character in buffer

ret                                             ; Return
show     ENDP
```

Note the following differences:

- Since ESP can be used as a base register, stack registers can be accessed directly without the stack setup required by previous processors. This assumes that ESP does not change inside the procedure.
- Values are loaded and zero extended in one step by using the MOVZX instruction (see Section 17.)
- EBX is used with scaling. In the previous example, scaling had to be done with a separate instruction.
- EAX and EBX are used instead of BX and SI. This saves some register swapping, since EAX can be used both for the result of the multiplication operation and as a base register.
- Immediate operands are used with the PUSH and IMUL instructions (described in Sections 17 and 18.) These enhancements were implemented with the 80186 processor, but they are rarely used since most programs have to be able to run on the 8088 and 8086. Since 80836 programs can never run on the earlier processors, there is no reason not to use enhanced 80186 instructions.

Section 17

Loading, Storing, and Moving Data

The 8086 family processors provide several instructions for loading, storing, or moving various kinds of data. Among the types of transferable data are variables, pointers, and flags. Data can be moved to and from registers, memory, ports, and the stack. This chapter explains the instructions for moving data from one location to another in the following sections:

- Transferring Data
- Converting between Data Sizes
- Loading Pointers
- Transferring Data to and from the Stack

Transferring Data

This subsection discusses moving data in the following ways:

- Copying Data
- Exchanging Data
- Looking Up Data
- Transferring Flags

Moving data is one of the most common tasks in assembly language programming. Data can be moved between registers or between memory and registers. Immediate data can be loaded into registers or into memory.

Copying Data

The MOV instruction is the most common method of moving data. This instruction can be thought of as a copy instruction, since it always copies the source operand to the destination operand. Immediately after a MOV instruction, the source and destination operands both contain the same value. The old value in the destination operand is destroyed. The syntax is:

MOV {*register* | *memory*},{*register* | *memory* | *immediate*}

The statements in the following example illustrate each type of memory move that can be done with a single instruction.

mov	ax,7	; Immediate to register
mov	mem,7	; Immediate to memory direct
mov	mem[bx],7	; Immediate to memory indirect
mov	mem,ds	; Segment register to memory
mov	mem,ax	; Register to memory direct
mov	mem[bx],ax	; Register to memory indirect
mov	ax,mem	; Memory direct to register
mov	ax,mem[bx]	; Memory indirect to register
mov	ds,mem	; Memory to segment register
mov	ax,bx	; Register to register
mov	ds,ax	; General register to segment register
mov	ax,ds	; Segment register to general register

The following example illustrates several common types of moves that require two instructions.

; Move immediate to segment register		
mov	ax,DGROUP	; Load immediate to general register
mov	ds,ax	; Store general register to segment register
; Move memory to memory		
mov	ax,mem1	; Load memory to general register
mov	mem2,ax	; Store general register to memory
; Move segment register to segment register		
mov	ax,ds	; Load segment register to general register
mov	es,ax	; Store general register to segment register

Exchanging Data

The XCHG (Exchange) instruction exchanges the data in the source and destination operands. Data can be exchanged between registers or between registers and memory. The syntax is:

XCHG {*register* | *memory*},{*register* | *memory*}

xchg	ax,bx	; Put AX in BX and BX in AX
xchg	memory,ax	; Put memory in AX and AX in
		; memory

Looking Up Data

The XLAT (Translate) instruction is used to load data from a table in memory. The instruction is useful for translating bytes from one coding system to another. The syntax is:

XLAT[B] [[*segment:*] *memory*]

The BX register must contain the address of the start of the table. By default the DS register contains the segment of the table, but a segment override can be used to specify a different segment. The operand need not be given except when specifying a segment override.

Before the XLAT instruction is called, the AL register should contain a value that points into the table (the start of the table is considered 0). After the instruction is called, AL will contain the table value pointed to. For example, if AL contains 7, the 8th byte of the table will be placed in AL register.

Note: *For compatibility with Intel 80386 mnemonics, MASM recognizes XLATB as a synonym for XLAT. In the Intel syntax, XLAT requires an operand; XLATB does not allow one. MASM never requires an operand, but always allows one.*

The following example looks up hexadecimal characters in a table in order to convert an 8 bit binary number to a string representing a hexadecimal number.

```
hex      ; Table of Hexadecimal digits
convert  DB      "0123456789ABCDEF"
key      DB      "You pressed the key with ASCII code"
         DB      ?,?,h,13,10,0
         .CODE
         .
         .
         .
push     ds      ; memory address of byte
push     OFFSET key ; to which char is to
               ; be returned
         .
int      ReadKbd ;
mov      bx,OFFSET hex ; Load table address
mov      ah,al    ; Save a copy in high byte
and      al,00001111b ; Mask out top character
xlat     ; Translate
mov      mov      key[1],al ; Store the character
mov      cl,12    ; Load shift count
shr      ax,cl    ; Shift high character
               ; into position
         .
xlat     ; Translate
mov      key,al   ; Store the character
mov      dx,OFFSET convert ; Load address
         .
push     ds      ; of message
push     dx      ;
call     zprint  ; Print it
```

Transferring Flags

The 8086 family processors provide instructions for loading and storing flags in the AH register.

LAHF
SAHF

The status of the lower byte of the flags register can be saved to the AH register with LAHF and then later restored with SAHF. If you need to save and restore the entire flags register, use PUSHF and POPF, as described in Saving Flags on the Stack.

SAHF is often used with a coprocessor to transfer coprocessor control flags to processor control flags. Section 21 explains and illustrates this technique.

Converting between Data Sizes

You can convert between data sizes by doing the following:

- Extending Signed Values
- Extending Unsigned Values
- Moving and Extending Values

Since moving data between registers of different sizes is illegal, you must take special steps if you need to extend a register value to a larger register or register pair. The procedure is different for signed and unsigned values. The processor cannot tell the difference between signed and unsigned numbers; the programmer has to understand this difference and program accordingly.

Extending Signed Values

The CBW (Convert Byte to Word) and CWD (Convert Word to Doubleword) instructions are provided to sign extend values. Sign extending means copying the sign bit of the unextended operand to all bits of the extended operand.

CBW

CWD

The CBW instruction converts an 8 bit signed value in AL to a 16 bit signed value in AX. The CWD instruction is similar except that it sign extends a 16 bit value in AX to a 32 bit value in the DX:AX register pair. Both instructions work only on values in the accumulator register.

```

mem8      .DATA          -5
mem16     DB             -5
          .CODE
          .
          .
          mov     al,mem8      ; Load 8 bit -5 (FBh)
          cbw                      ; Convert to 16 bit -5 (FFFBh) in
                                ;     AX
          mov     ax,mem16     ; Load 16 bit -5 (FFFBh)
          cwd                      ; Convert to 32 bit -5
                                ;     (FFFF:FFFBh)
                                ;     in DX:AX

```

The 80386 processor provides additional conversion instructions for 32 bit signed values.

CWDE

CDQ

The CWDE (Convert Word to Doubleword Extended) instruction converts a signed 16 bit value in AX to a signed 32 bit signed value in EAX. The CDQ (Convert Doubleword to Quadword) instruction converts a 32 bit signed value in EAX to a signed 64 bit value in the EDX:EAX register pair as shown in the following example:

```
.DATA
mem16      DW -5
mem32      DD -5
.CODE
.
.
.
mov        ax,mem16      ; Load 16 bit -5 (FFFBh)
cwde       ; Convert to 32 bit -5 (FFFFFFFBh)
           ;      in EAX
mov        eax,mem32     ; Load 32 bit -5 (FFFFFFFBh)
cdq        ; Convert to 64 bit -5
           ;      (FFFFFFFF:FFFFFFFBh) in
           ;      EDX:EAX
```

Extending Unsigned Values

To extend unsigned numbers, set the value of the upper register to 0.

```
.DATA
mem8       DB 251
mem16      DW 251
.CODE
.
.
.
mov        al,mem8       ; Load 251 (FBh) from 8 bit memory
xor        ah,ah         ; Zero upper half (AH)

mov        ax,mem16      ; Load 251 (FBh) from 16 bit
                        ;      memory
xor        dx,dx         ; Zero upper half (DX)
```

Moving and Extending Values

The 80386 processor provides instructions that move and extend a value to a larger data size in a single step. The same thing can be done in two steps with earlier processors, but the new 80386 instructions are faster.

MOVSX *register*,{*register* | *memory*}

MOVZX *register*,{*register* | *memory*}

MOVSX moves a signed value into a register and sign extends it.

MOVZX moves an unsigned value into a register and zero extends it.

; Enhanced 80386 instructions

```
movzx    dx,bl                ; Load unsigned 8 bit value into
                               ; 16 bit register and zero
                               ; extend
```

; Equivalent to these 80286 instructions

```
mov      dl,bl                ; Load 8 bit unsigned value
xor      dh,dh                ; Clear the top of register
```

; Enhanced 80386 instructions

```
movsx    dx,bl                ; Load unsigned 8 bit value into
                               ; 16 bit register and sign
                               ; extend
```

; Equivalent to these 80286 instructions

```
mov      al,bl                ; Load 8 bit unsigned value to AL
cbw                               ; Sign extend to AX
mov      dx,ax                ; Copy to 16 bit register
```

Loading Pointers

The 8086 family processors provide several instructions for loading pointer values into registers or register pairs. They can be used to load either near or far pointers. They are divided into two groups:

- Near Pointers
- Far Pointers

Loading Near Pointers

The LEA instruction loads a near pointer into a specified register. The syntax is:

LEA register,memory

The destination register may be any general purpose register. The source operand may be any memory operand. The effective address of the source operand is placed in the destination register.

The LEA instruction can be used to calculate the effective address of a direct memory operand, but this is usually not efficient, since the address of a direct memory operand is a constant known at assembly time. For example, the following statements have the same effect, but the second version is faster:

```
lea    dx,string           ; Load effective address –
                           ;      slow
mov     dx,OFFSET string   ; Load offset – fast
```

The LEA instruction is more useful for calculating the address of indirect memory operands:

```
lea dx,string[si] ; Load effective address
```

Scaling of indirect memory operands gives the LEA instruction some interesting side effects with the 80386 processor. (Scaling is explained in Section 16). By using a 32 bit value as both the index and the base register in an indirect memory operand, you can multiply by the constants 2, 3, 4, 5, 8, and 9 more quickly than you could by using the MUL instruction.

```
lea     ebx,[eax*2]         ; EBX = 2 * EAX
lea     ebx,[eax*2+eax]     ; EBX = 3 * EAX
lea     ebx,[eax*4]         ; EBX = 4 * EAX
lea     ebx,[eax*4+eax]     ; EBX = 5 * EAX
lea     ebx,[eax*8]         ; EBX = 8 * EAX
lea     ebx,[eax*8+eax]     ; EBX = 9 * EAX
```

Multiplication by constants can also sometimes be made faster by using shift instructions, as described in Section 18.

Loading Far Pointers

The LDS and LES instructions load far pointers.

LDS *register,memory*

LES *register,memory*

The memory address being pointed to is specified in the source operand, and the register where the offset will be stored is specified in the destination operand.

The address must be stored in memory with the segment in the upper word and the offset in the lower word. The segment register where the segment will be stored is specified in the instruction name. For example, LDS puts the segment in DS, and LES puts the segment in ES. These instructions are often used with string instructions, as explained in Section 20.

```
                .DATA
string          DB          "This is a string."
fpstring        DD          string          ; Far pointer to string
pointers        DD          100 DUP (?)
                .CODE
                :
                :
                les         di,fpstring      ; Put address in ES:DI
                                   ; pair
                lds         si,pointers[bx]  ; Put address in DS:SI
                                   ; pair
```

The 80386 processor has additional instructions for loading far pointers. These instructions are exactly like LDS and LES, except for the segment register in which they put the segment address.

LSS *register,memory*

LFS *register,memory*

LGS *register,memory*

The LSS, LFS, and LGS instructions load the segment address into SS, FS, and GS respectively.

```
.386                                ; .386 first for 32 bit
                                    ; mode
.MODEL    large
.DATA
string    DB    'This is a string.'
fpstring  DF string                ; Far pointer to string
.CODE
.
.
.
lgs edi,fpstring                    ; Put address in GS:EDI
                                    ; pair
```

Transferring Data to and from the Stack

A stack is an area of memory for storing temporary data. Unlike other segments in which data is stored starting from low memory, data on the stack is stored in reverse order starting from high memory. The stack can be manipulated in the following ways:

- Pushing and Popping
- Saving Flags on the Stack
- Saving All Registers on the Stack

Initially, the stack is an uninitialized segment of a finite size. As data is added to the stack at run time, the stack grows downward from high memory to low memory. When items are removed from the stack, it shrinks upward from low memory to high memory.

The stack has several purposes in the 8086 family processors. The CALL, INT, RET, and IRET instructions automatically use the stack to store the calling addresses of procedures and interrupts (see Section 19). You can also use the PUSH and POP instructions and their variations to store values on the stack.

Pushing and Popping

In 8086 family processors, the SP (stack pointer) register always points to the current location in the stack. The PUSH and POP instructions use the SP register to keep track of the current position in the stack.

PUSH { <i>register</i> <i>memory</i> }	
POP { <i>register</i> <i>memory</i> }	.
PUSH <i>immediate</i>	@(80186–80386 only)

Note: The 8088 and 8086 processors differ from later Intel processors in how they push and pop the SP register. If you give the statement `push sp` with the 8088 or 8086, the word pushed will be the word in SP after the push operation. The same statement under the 80186, 80286, or 80386 processor pushes the word in SP before the push operation.

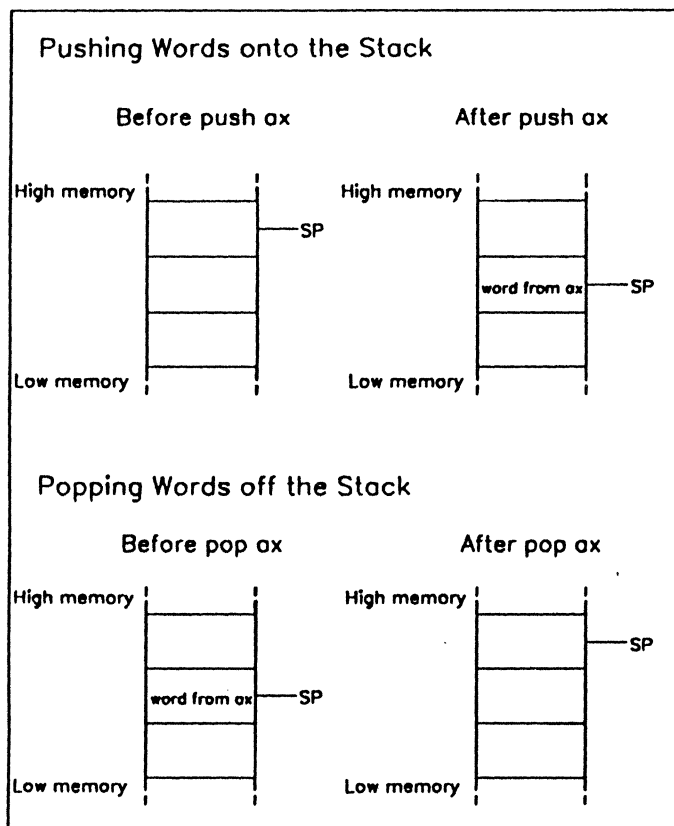


Figure 17-1. Stack Status after Pushes and Pops

The PUSH and POP instructions are almost always used in pairs. Words are popped off the stack in reverse order from the order in which they are pushed onto the stack. You should normally do the same number of pops as pushes to return the stack to its original status. However, it is possible to return the stack to its original status by subtracting the correct number of words from the SP register.

Values on the stack can be accessed by using indirect memory operands with BP as the base register.

```

mov      bp,sp                ; Set stack frame
push     ax                   ; Push first; SP = BP + 2
push     bx                   ; Push second; SP = BP + 4
push     cx                   ; Push third; SP = BP + 6
.
.
mov      ax,[bp+6]             ; Put third in AX
mov      bx,[bp+4]             ; Put second in BX
mov      cx,[bp+2]             ; Put first in CX
.
.
add      sp,6                  ; Restore stack pointer
                                   ; two bytes per push

```

Starting with the 80186, the PUSH instruction can be given with an immediate operand. For example, the following statement is legal on the 80186, 80286, and 80386 processors:

```

push     7                    ; 3 clocks on 80286

```

This statement is faster than the following equivalent statements, which are required on the 8088 or 8086:

```

mov      ax,7                  ; 2 clocks on 80286
push     ax                    ; 3 clocks on 80286

```

When a PUSH or POP instruction is used in a 32 bit code segment (one with USE32 use type), the value transferred is a four byte value. A warning message will be generated if you try to push a 16 bit value in a 32 bit segment or a 32 bit value in a 16 bit segment.

Using the Stack

The stack can be used to store temporary data. For example, in the Microsoft calling convention, the stack is used to pass arguments to a procedure. The arguments are pushed onto the stack before the call. The procedure retrieves and uses them. Then the stack is restored to its original position at the end of the procedure. The stack can also be used to store variables that are local to a procedure. Both these techniques are discussed in Section 19.

Another common use of the stack is to store temporary data when there are no free registers available or when a particular register must hold more than one value. For example, the CX register usually holds the count for loops. If two loops are nested, the outer count is loaded into CX at the start. When the inner loop starts, the outer count is pushed onto the stack and the inner count loaded into CX. When the inner loop finishes, the originally count is popped back into CX.

```
mov     cx,10                ; Load outer loop counter
outer:
.
.                ; Start outer loop task
.
push cx                    ; Save outer loop value
mov cx,20                ; Load inner loop counter
inner:
.
.                ; Do inner loop task
.
loop inner
pop cx                    ; Restore outer loop counter

.
.                ; Continue outer loop task
.
loop outer
```

Saving Flags on the Stack

Flags can be pushed and popped onto the stack using the PUSHF and POPF instructions.

PUSHF
POPF

These instructions are sometimes used to save the status of flags before a procedure call and then to restore the same status after the procedure. They can also be used within a procedure to save and restore the flag status of the caller.

```
pushf
call    systask
popf
```

When used from a 32 bit code segment, the PUSHF and POPF instructions do not automatically transfer 32 bit values. You must append the letter D (for doubleword) to the instruction name. Thus the 32 bit versions of these instructions are PUSHFD and POPFD.

Saving All Registers on the Stack

Starting with the 80186 processor, the PUSHA and POPA instructions were implemented to push or pop all the general purpose registers with one instruction.

PUSHA
POPA

These instructions can be used to save the status of all registers before a procedure call and then to restore them after the return. Using PUSHA and POPA instructions is significantly faster and takes fewer bytes of code than pushing and popping each register individually.

The registers are pushed in the following order: AX, CX, DX, BX, SP, BP, SI, and DI. The SP word pushed is the value before the first register is pushed. The registers are popped in the opposite order.

```
pusha
call    systask
popa
```

When used from a 32 bit code segment, the PUSH, PUSHA, POP, and POPA instructions do not automatically transfer 32 bit values. You must append the letter D (for doubleword) to the instruction name. Thus the 32 bit versions of these instructions are PUSH32, PUSHAD, POP32, and POPAD.

Transferring Data to and from Ports

Ports are the gateways between hardware devices and the processor. Each port has a unique number through which it can be accessed. Ports can be used for low level communication with devices such as disks, the video display, or the keyboard. The OUT instruction is used to send data to a port; the IN instruction receives data from a port.

IN *accumulator*,{*portnumber* | DX}
OUT {*portnumber* | DX},*accumulator*

When using the IN and OUT instructions, the number of the port can either be an 8 bit immediate value or the DX register. You must use DX for ports with a number higher than 256. The value to be received from the port must be in the accumulator register (AX for word values or AL for byte values).

When using the IN instruction, the number of the port is given as the source operand and the value to be transferred to the port is the destination operand. When using the OUT instruction, the number of the port is given as the destination operand and the value to be sent to the port is the source operand.

Ports are more often used in systems programming. Since systems programming is beyond the scope of this manual and since ports differ depending on hardware, the IN and OUT instructions are not explained in detail here.

Under protected mode operating systems, IN and OUT are privileged instructions and can only be used in privileged mode.

Starting with the 80186 processor, instructions were implemented to send strings of data to and from ports. The instructions are INS, INSB, INSW, OUTS, OUTSB, and OUTSW. The operation of these instructions is much like the operation of other string instructions. They are discussed in Section 20.

Section 18

Doing Arithmetic and Bit Manipulations

The 8086 family processors provide instructions for doing calculations on byte, word, and doubleword values. Operations include addition, subtraction, multiplication, and division. You can also do calculations at the bit level. This includes the AND, OR, XOR, and NOT logical operations. Bits can also be shifted or rotated to the right or left. This chapter tells you how to use the instructions that do calculations on numbers and bits.

Adding

The ADD, ADC, and INC instructions are used for adding and incrementing values. The syntax is:

```
ADD {register | memory},{register | memory | immediate}  
ADC {register | memory},{register | memory | immediate}  
INC {register | memory}
```

These instructions can work directly on 8 bit or 16 bit values (32 bit values on the 80386). They can be also be used in combination to do calculations on values that are too large to be held in a single register (such as 32 bit values on the 8086 or 64 bit values on the 80386). When used with AAA and DAA, they can be used to do calculations on BCD numbers, as described in Section 18.

Adding Values Directly

The ADD and INC instructions are used for adding to values in registers or memory. The INC instruction takes a single register or memory operand. The value of the operand is incremented. The value is treated as an unsigned integer, so the carry flag is not updated for signed carries.

The ADD instruction adds values given in source and destination operands. The destination can be either a register or a memory operand. Its contents will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. Since memory to memory operations are never allowed, the source and destination operands can never both be memory operands.

The result of the operation is stored in the source operand. The operands can be either 8 bit or 16 bit (32 bit on the 80386), but both must be the same size.

An addition operation can be interpreted as addition of either signed numbers or unsigned numbers. It is the programmer's responsibility to decide how the addition should be interpreted and to take appropriate action if the sum is too large for the destination operand. When an addition overflows the possible range for signed numbers, the overflow flag is set. When an addition overflows the range for unsigned numbers, the carry flag is set.

There are two ways to take action on an overflow: you can use the JO or JNO instruction to direct program flow to or around instructions that handle the overflow (see Section 19). You can also use the INTO instruction to trigger the overflow interrupt (interrupt 4) if the overflow flag is set. This requires writing an interrupt handler for interrupt 4. Section 19 gives a sample of an overflow interrupt handler.

The following example shows 8 bit addition. When the sum exceeds 127, the overflow flag is set. A JO (Jump on Overflow) or INTO (Interrupt on Overflow) instruction at this point could transfer control to error recovery statements. When the sum exceeds 255, the carry flag is set. A JC (Jump on Carry) instruction at this point could transfer control to error recovery statements.

mem8	.DATA								
	DB	39							
	.CODE								
	.								
	.								
	mov	al,26		;	Start with register	unsigned	signed		
	inc	al		;	Increment	26	26		
	add	al,76		;	Add immediate	1	1		
				;		+ 76	76		
				;					
				;		103	103		
	add	al,mem8		;	Add memory	+ 39	39		
				;					
	mov	ah,al		;	Copy to AH	142	-114		
				;			+overflow		
	add	al,ah		;	Add register	142			
				;					
				;					
				;			28+carry		

Adding Values in Multiple Registers

The ADC (Add with Carry) instruction makes it possible to add numbers larger than can be held in a single register.

The ADC instruction adds two numbers in the same fashion as the ADD instruction, except that the value of the carry flag is included in the addition. If a previous calculation has set the carry flag, then 1 will be added to the sum of the numbers. If the carry flag is not set, the ADC instruction has the same effect as the ADD instruction.

When adding numbers in multiple registers, the carry flag should be ignored for the least significant portion, but taken into account for the more significant portion. This can be done by using the ADD instruction for the least significant portion and the ADC instruction for more significant portions.

You can add and carry repeatedly inside a loop for calculations that require more than two registers. Use the ADC instruction in each iteration, but turn off the carry flag with the CLC (Clear Carry Flag) instruction before entering the loop so that it will not be used for the first iteration. You could also do the first add outside the loop. As follows:

```
mem32      .DATA
            DD      316423
            .CODE
            .
            .
            mov     ax,43981                ; Load immediate 43981
            xor     dx,dx                   ; into DX:AX
            add     ax,WORD PTR mem32[0]    ; Add to both +316423
            adc     dx,WORD PTR mem32[2]    ; memory words
                                           ; Result in DX:AX 360404
```

Subtracting

The SUB, SBB, DEC, and NEG instructions are used for subtracting and decrementing values. The syntax is:

```
SUB {register | memory},{register | memory | immediate}
SBB {register | memory},{register | memory | immediate}
DEC {register | memory}
NEG {register | memory}
```

These instructions can work directly on 8 bit or 16 bit values (32 bit values on the 80386). They can be also be used in combination to do calculations on values too large to be held in a single register (such as 32 bit values on the 8086 or 64 bit values on the 80386). When used with AAA and DAA, they can be used to do calculations on BCD numbers, as described in Calculating with Binary Coded Decimals.

Subtracting Values Directly

The SUB and DEC instructions are used for subtracting from values in registers or memory. A related instruction, NEG (Negate), reverses the sign of a number.

The DEC instruction takes a single register or memory operand. The value of the operand is decremented. The value is treated as an unsigned integer, so the carry flag is not updated for signed borrows.

The NEG instruction takes a single register or memory operand. The sign of the value of the operand is reversed. The NEG instruction should only be used on signed numbers.

The SUB instruction subtracts the values given in the source operand from the value of the destination operand. The destination can be either a register or a memory operand. It will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. It will not be destroyed by the operation. Since memory to memory operations are never allowed, the source and destination operands cannot both be memory operands.

The result of the operation is stored in the source operand. The operands can be either 8 bit or 16 bit (32 bit on the 80386), but both must be the same size.

A subtraction operation can be interpreted as subtraction of either signed numbers or of unsigned numbers. It is the programmer's responsibility to decide how the subtraction should be interpreted and to take appropriate action if the result is too small for the destination operand. When a subtraction overflows the possible range for signed numbers, the carry flag is set. When a subtraction underflows the range for unsigned numbers (becomes negative), the sign flag is set.

The following example shows 8 bit subtraction. When the result goes below 0, the sign flag is set. A JS (Jump on Sign) instruction at this point could transfer control to error recovery statements. When the result goes below -128, the carry flag is set. A JC (Jump on Carry) instruction at this point could transfer control to error recovery statements.

mem8	.DATA			
	DB	122		
	.CODE			
	.			
	.			
	.			
	mov	al,95	; Load register	signed 95 unsigned 95
	dec	al	; Decrement	— 1 — 1
	sub	al,23	; Subtract immediate	— 23 — 23
	sub	al,mem8	; Subtract memory	— 122 — 122
	mov	ah,119	; Load register	119
	sub	al,ah	; and subtract	— 51
				86
				+ overflow

Subtracting with Values in Multiple Registers

The SBB (Subtract with Borrow) instruction makes it possible to subtract from numbers larger than can be held in a single register.

The SBB instruction subtracts two numbers in the same fashion as the SUB instruction except that the value of the carry flag is included in the subtraction. If a previous calculation has set the carry flag, then 1 will be subtracted from the result. If the carry flag is not set, the SBB instruction has the same effect as the SUB instruction.

When subtracting numbers in multiple registers, the carry flag should be ignored for the least significant portion, but taken into account for the more significant portion. This can be done by using the SUB instruction for the least significant portion and the SBB instruction for more significant portions.

You can subtract and borrow repeatedly inside a loop for calculations that require more than two registers. Use the SBB instruction in each iteration, but turn off the carry flag with the CLC (Clear Carry Flag) instruction before entering the loop so that it will not be used for the first iteration. You could also do the first subtraction outside the loop as in the following example:

```
mem32a      .DATA      DD      316423
mem32b      DD      156739
             .CODE
             .
             .
             .
             mov     ax,WORD PTR mem32a[0]      ;Load mem32      316423
             mov     dx,WORD PTR mem32a[2]      ; into DX:AX
             sub     ax,WORD PTR mem32b[0]      ;Subtract low    156739
             sbb     dx,WORD PTR mem32b[2]      ;then high
                                                    ;Result in DX:AX 159684
```

Multiplying

The MUL and IMUL instructions are used to multiply numbers. The MUL instruction should be used for unsigned numbers; the IMUL instruction should be used for signed numbers. This is the only difference between the two. The syntax is:

```
MUL {register | memory}
IMUL {register | memory}
```

The multiply instructions require that one of the factors be in the accumulator register (AL for 8 bit numbers, AX for 16 bit numbers, or EAX for 32 bit numbers). This register is implied; it should not be specified in the source code. Its contents will be destroyed by the operation.

The other factor to be multiplied must be specified in a single register or memory operand. The operand will not be destroyed by the operation, unless it is DX, AH, or AL.

Note that multiplying two 8 bit numbers will produce a 16 bit number. If the product is a 16 bit number, it will be placed in AX and the overflow and carry flags will be set.

Similarly, multiplying two 16 bit numbers will produce a 32 bit number in the DX:AX register pair. If the product is a 32 bit number, the least significant bits will be in AX, the most significant bits will be in DX, and the overflow and carry flags will be set. (The 80386 handles 64 bit products in the same way in the EDX:EAX register pair.)

Note: *Multiplication is one of the slower operations on 8086 family processors (especially the 8086 and 8088). Multiplying by certain common constants is often faster when done by shifting bits (see *Multiplying and Dividing by Constants*) or by using 80386 scaling (see Section 17).*

The following example shows both an 8 bit and 16 bit multiply:

mem16	.DATA		
	DW	-30000	
	.CODE		
	.		
	.		
		; 8-bit unsigned multiply	
mov	al,23	; Load AL	23
mov	bl,24	; Load BL	* 24
mul	bl	; Multiply BL	
		; Product in AX	552
		; overflow and carry set	
		; 16-bit signed multiply	
mov	ax,50	; Load AX	50
			-30000
imul	mem16	; Multiply memory	
		; Product in DX:AX	-1500000
		; overflow and carry set	

Starting with the 80186, the IMUL instruction has two additional syntaxes that allow for 16 bit multiples that produce a 16 bit product. (These instructions can be extended to 32 bits on the 80386.) The syntax is:

IMUL register16,immediate

IMUL register16,memory16,immediate

You can specify a 16 bit immediate value as the source instruction and a word register as the destination operand. The product appears in the destination operand. The 16 bit result will be placed in the destination operand. If the product is too large to fit in 16 bits, the carry and overflow flags will be set. In this context, IMUL can be used for either signed or unsigned multiplication, since the 16 bit product is the same.

You can also specify three operands for IMUL. The first operand must be a 16 bit register operand, the second a 16 bit memory operand, and the third a 16 bit immediate operand. The second and third operands are multiplied and the product stored in the first operand.

With both these syntaxes, the carry and overflow flags will be set if the product is too large to fit in 16 bits. The IMUL instruction with multiple operands can be used for either signed or unsigned multiplication, since the 16 bit product is the same in either case. If you need to get a 32 bit result, you must use the single operand version of MUL or IMUL. As in the following example.

```
imul dx,456      ; Multiply DX times 456
imul ax,[bx],6   ; Multiply the value pointed
                  ; to by BX times and put
                  ; the result in AX
```

On the 80386, the IMUL instruction has an additional instruction that allows multiplication of a register value by a register or memory value. The syntax is:

IMUL *register*,{*register* | *memory*}

As the following example illustrates, the destination can be any 16 bit or 32 bit register. The source must be the same size as the destination. As follows:

```
imul dx,ax       ; Multiply DX times AX
imul ax,[bx]     ; Multiply AX by the value pointed
                  ; to by BX
```

Dividing

The DIV and IDIV instructions are used to divide integers. Both a quotient and a remainder are returned. The DIV instruction should be used for unsigned integers; the IDIV instruction should be used for signed integers. This is the only difference between the two. The syntax is:

DIV {*register* | *memory*}
IDIV {*register* | *memory*}

To divide a 16 bit number by an 8 bit number, put the number to be divided (the dividend) in the AX register. The contents of this register will be destroyed by the operation. Specify the dividing number (the divisor) in any 8 bit memory or register operand (except AL or AH). This operand will not be changed by the operation. After the multiplication, the result (quotient) will be in AL and the remainder will be in AH.

To divide a 32 bit number by a 16 bit number, put the dividend in the DX:AX register pair. The least significant bits go in AX. The contents of these registers will be destroyed by the operation. Specify the divisor in any 16 bit memory or register operand (except AX or DX). This operand will not be changed by the operation. After the division, the quotient will be in AX and the remainder will be in DX. (The 80386 handles 64 bit division in the same way by using the EDX:EAX register pair.)

To divide a 16 bit number by a 16 bit number, you must first sign extend or zero extend (see Section 17) the dividend to 32 bits; then divide as described above. You cannot divide a 32 bit number by another 32 bit number (except on the 80386).

If division by zero is specified, or if the quotient exceeds the capacity of its register (AL or AX), the processor automatically generates an interrupt 0. By default, the program terminates. This problem can be handled in two ways: you can check the divisor before division and go to an error routine if you can determine it to be invalid, or you can write your own interrupt routine to replace the processor's interrupt 0 routine. See Section 19 for more information in interrupts.

Note: *Division is one of the slower operations on 8086 family processors (especially the 8086 and 8088). Dividing by common constants that are powers of two is often faster when done by shifting bits, as described in Multiplying and Dividing by Constants.*

The following example illustrates the division instruction:

```

mem16      .DATA      -2000
mem32      DD          500000
            .CODE
            .
            .          ; Divide 16 bit unsigned by 8 bit
            .
            mov     ax,700      ; Load dividend      700
            mov     bl,36      ; Load divisor      DIV 36
            div     bl          ; Divide BL
            ; Quotient in AL      19
            ; Remainder in AH    16
            .
            .          ; Divide 32-bit signed by 16-bit
            mov     ax,WORD PTR mem32[0] ; Load into DX:AX
            mov     dx,WORD PTR mem32[2] ;
            idiv    mem16      ; Divide memory      DIV 500000
            ; Quotient in AX      -2000
            ; Remainder in DX    0
            .
            .          ; Divide 16 bit signed by 16 bit
            mov     ax,WORD PTR mem16    ; Load into AX      -2000
            cwd     ; Extend to DX:AX
            mov     bx,-421              ; Divide by BX      DIV -421
            idiv    bx                  ; Quotient in AX      4
            ; Remainder in DX      -316

```

Calculating with Binary Coded Decimals

The 8086 family processors provide several instructions for adjusting BCD numbers. The BCD format is seldom used for applications programming in assembly language. Programmers who wish to use BCD numbers usually use a high level language. However, BCD instructions are used to develop compilers, function libraries, and other systems tools.

Since systems programming is beyond the scope of this manual, this section provides only a brief overview of calculations on the two kinds of BCD numbers, unpacked and packed.

Note: *Intel mnemonics use the term ASCII to refer to unpacked BCD numbers and decimal to refer to packed BCD numbers. Thus AAA (ASCII Adjust for Addition) adjusts unpacked numbers, while DAA (Decimal Adjust for Addition) adjusts packed numbers.*

Unpacked BCD Numbers

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower four bits of each byte. The 8086 family processors provide instructions for adjusting unpacked values with the four arithmetic operations addition, subtraction, multiplication, and division.

To do arithmetic on unpacked BCD numbers, you must do the 8 bit arithmetic calculations on each digit separately. The result should always be in the AL register. After each operation, use the corresponding BCD instruction to adjust the result. The ASCII adjust instructions do not take an operand. They always work on the value in the AL register.

When a calculation using two one digit values produces a two digit result, the ASCII adjust instructions put the first digit in AL and the second in AH. If the digit in AL needs to carry to or borrow from the digit in AH, the carry and auxiliary carry flags are set.

The four ASCII adjust instructions are described below:

AAA adjusts after an addition operation. For example, to add 9 and 3, put 9 in AL and 3 in BL. Then use the following lines to add them:

```
mov    ax,9      ; Load 9
mov    bx,3      ; and 3 as unpacked BCD.
add    al,bl     ; Add 09h and 03h to get 0Ch
aaa                    ; Adjust 0Ch in AL to 02h,
                    ; increment AH to 01h, set carry
                    ; Result 12 unpacked BCD in AX
```

AAS Adjusts after a subtraction operation. For example, to subtract 4 from 13, put 13 (103h) in AX and 4 in BL. Then use the following lines to subtract them:

```
mov    ax,103h   ; Load 13
mov    bx,4      ; and 4 as unpacked BCD
sub    al,bl     ; Subtract 4 from 3 to get FFh (-1)
aas                    ; Adjust 0FFh in AL to 9,
                    ; decrement AH to 0, set carry
                    ; Result 9 unpacked BCD in AX
```

AAM Adjusts after a multiplication operation. Always use MUL, not IMUL. For example, to multiply 9 times 3, put 9 in AH and 3 in AL. Then use the following lines to multiply them:

```

mov     ax,903h      ; Load 9 and 3 as unpacked BCD
mul     ah           ; Multiply 9 and 3 to get 18h
aam     ; Adjust 18h in AL
                ; to get 27 unpacked BCD in AX

```

AAD Adjusts before a division operation. Unlike other BCD instructions, this one converts a BCD value to a binary value before the operation. After the operation, the quotient must still be adjusted by using AAM. For example, to divide 25 by 2, put 25 in AX in unpacked BCD format: 2 in AH and 5 in AL. Put 2 in BL. Then use the following lines to divide them:

```

mov     ax,205h      ; Load 25
mov     bl,2         ; and 2 as unpacked BCD
aad     ; Adjust 0205h in AX
                ; to get 19h in AX
div     bl           ; Divide by 2 to get
                ; quotient 0Ch in AL
                ; remainder 1 in AH
aam     ; Adjust 0Ch in AL
                ; to 12 unpacked BCD in AX
                ; (remainder destroyed)

```

Notice that the remainder is lost. If you need the remainder, save it in another register before adjusting the quotient. Then move it back to AL and adjust if necessary.

Multidigit BCD numbers are usually processed in loops. Each digit is processed and adjusted in turn. In addition to their use for processing unpacked BCD numbers, the ASCII adjust instructions can be used in routines that convert between different number bases. The following example demonstrates the use of ASCII adjust instructions:

```

mov     al,79        ; Load 79 (04Fh)
aam     ; Adjust to BCD (0709h)
add     ah,48        ; Adjust to ASCII characters
add     al,48        ; (3739h)
xchg    al,ah        ; trade for most sig byte
push    ax           ;
call    PutChar      ;
mov     al,ah        ; least significant byte
call    PutChar      ;

```

The example converts an 8 bit binary number to hexadecimal and displays it on the screen. The routine could be enhanced to handle large numbers.

Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper four bits and one in the lower four bits. The 8086 family processors provide instructions for adjusting packed BCD numbers after addition and subtraction. You must write your own routines to adjust for multiplication and division.

To do arithmetic on packed BCD numbers, you must do the eight bit arithmetic calculations on each byte separately. The result should always be in the AL register. After each operation, use the corresponding BCD instruction to adjust the result. The decimal adjust instructions do not take an operand. They always work on the value in the AL register.

Unlike the ASCII adjust instructions, the decimal adjust instructions never affect AH. The auxiliary carry flag is set if the digit in the lower four bits carries to or borrows from the digit in the upper four bits. The carry flag is set if the digit in the upper four bits needs to carry to or borrow from another byte.

The decimal adjust instructions are described below:

DAA Adjusts after an addition operation. For example, to add 88 and 33, put 88 in AH and 33 in AL in packed BCD format. Then use the following lines to add them:

```
mov    ax,8833h    ; Load 88 and 33 as packed BCD
add    al,ah        ; Add 88 and 33 to get 0BBh
daa    ; Adjust 0BBh to 121 packed BCD:
        ; 1 in carry and 21 in AL
```

DAS Adjusts after a subtraction operation. For example, to subtract 38 from 83, put 83 in AL and 38 in AH in packed BCD format. Then use the following lines to subtract them:

```
mov    ax,3883h    ; Load 83 and 38 as packed BCD
sub    al,ah        ; Subtract 38 from 83 to get 04Bh
das    ; Adjust 04Bh to 45 packed BCD:
        ; 0 in carry and 45 in AL
```

Multidigit BCD numbers are usually processed in loops. Each byte is processed and adjusted in turn.

Doing Logical Bit Manipulations

The logical instructions do Boolean operations on individual bits. The AND, OR, XOR, and NOT operations are supported by the 8086 family instructions.

AND compares two bits and sets the result if both bits are set. OR compares two bits and sets the result if either bit is set. XOR compares two bits and sets the result if the bits are different. NOT reverses a single bit. Table 18–1 shows a truth table for the logical operations.

Table 18–1. Values Returned by Logical Operations

X	Y	NOT X	X AND Y	X OR Y	X XOR Y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

The syntax of the AND, OR, and XOR instructions are the same. The only difference is the operation performed. For all instructions, the target value to be changed by the operation is placed in one operand. A mask showing the positions of bits to be changed is placed in the other operand. The format of the mask differs for each logical instruction. The destination operand can be register or memory. The source operand can be register, memory, or immediate. However, the source and destination operands cannot both be memory.

Either of the values can be in either operand. However, the source operand will be unchanged by the operation, while the destination operand will be destroyed by it. Your choice of operands depends on whether you want to save a copy of the mask or of the target value.

Note: *The logical instructions should not be confused with the logical operators. They specify completely different behavior. The instructions control run time bit calculations. The operators control assembly time bit calculations. Although the instructions and operators have the same name, the assembler can distinguish them from context.*

AND Operations

The AND instruction does an AND operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits. The syntax is:

```
AND {register | memory},{register | memory | immediate}
```

As shown in the following example the AND instruction can be used to clear the value of specific bits regardless of their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 0 for any bit positions you want to clear and 1 for any bit positions you want to remain unchanged. For example:

```
mov    ax,035h      ; Load value          00110101
and    ax,0FBh      ; Mask off bit 2      AND  11111011
;
; Value is now 31h          00110001
and    ax,0F8h      ; Mask off bits 2,1,0  AND  11111000
;
; Value is now 30h          00110000
```

The following example illustrates how to use the AND instruction to convert a character to uppercase. If the character is already uppercase, the AND instruction has no effect, since bit 5 is always clear in uppercase letters. If the character is lowercase, clearing bit 5 converts it to uppercase.

```
char db 'a'
.
.
mov    al,char
and    al,11011111b ; Convert to uppercase by
; clearing bit 5
cmp    al,'A'       ; Is it A?
je     yes          ; If so, do Yes stuff
.                  ; else do No stuff
.
yes: .
```

OR Operations

The OR instruction does an OR operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits. The syntax is:

OR {*register* | *memory*},{*register* | *memory* | *immediate*}

As shown in the following example, the OR instruction can be used to set the value of specific bits regardless of their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 1 for any bit positions you want to set and 0 for any bit positions you want to remain unchanged. As follows:

mov	ax,035h	; Move value to register	00110101
or	ax,08h	; Mask on bit 3 OR	00001000
		; Value is now 3Dh	00111101
or	ax,07h	; Mask on bits 2,1,0 OR	00000111
		; Value is now 3Fh	00111111

Another common use for OR is to compare an operand to 0. For example:

or	bx,bx	; Compare to 0
		; 2 bytes, 2 clocks on 8088
jg	positive	; BX is positive
jl	negative	; BX is negative
		; BX is zero

The first statement has the same effect as the following statement, but is faster and smaller:

cmp	bx,0	; 3 bytes, 3 clocks on 8088
-----	------	-----------------------------

XOR Operations

The XOR (Exclusive OR) instruction does an XOR operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits. The syntax is:

XOR {*register* | *memory*},{*register* | *memory* | *immediate*}

As shown in the following example, XOR instruction can be used to toggle the value of specific bits (reverse them from their current settings). To do this, put the target value in one operand and a mask of the bits you want to toggle in the other. The bits of the mask should be 1 for any bit positions you want to toggle and 0 for any bit positions you want to remain unchanged.

```
mov    ax,035h      ; Move value to register      00110101
xor    ax,08h        ; Mask on bit 3              XOR      00001000
                                     ;
                                     ; Value is now 3Dh      00111101
xor    ax,07h        ; Mask on bits 2,1,0 XOR      00000111
                                     ;
                                     ; Value is now 3Ah      00111010
```

Another common use for the XOR instruction is to set a register to 0. For example:

```
xor    cx,cx          ; 2 bytes, 3 clocks on 8088
```

This sets the CX register to 0. When the identical operands are XORed, each bit cancels itself, producing 0. The statement

```
mov    cx,0           ; 3 bytes, 4 clocks on 8088
```

is the obvious way of doing this, but it is larger and slower. The statement

```
sub    cx,cx          ; 2 bytes, 3 clocks on 8088
```

is also smaller than the MOV version. The only advantage of using MOV is that it does not affect any flags.

NOT Operations

The NOT instruction does a NOT operation on the bits of a single operand. It is used to toggle the value of all bits at once. The syntax is:

```
NOT {register | memory}
```

As shown in the following example, the NOT instruction is often used to reverse the sense of a bit mask from masking certain bits on to masking them off. Use the NOT instruction if the value of the mask is not known until run time; use the NOT operator (see Section 11) if the mask is a constant.

	.DATA			
masker	DB	00010000b	; Value may change at run time	
	.CODE			
	.			
	.			
	.			
mov	ax,0D743h	; Load 0D7h to AH; 43h to AL		01000011
or	al,masker	; Turn on bit 4 in AL	OR	00010000
		; Result is 53h		01010011
not	masker	; Reverse sense of mask		11101111
and	ah,masker	; Turn off bit 4 in AH AND		11010111
		; Result is 0C7h		11000111

Scanning for Set Bits

The 80386 processor has instructions for scanning bits to find the first or last set bit in a register value. These instructions can be used to find the position of a set bit in a mask or other value. They can also check to see if a register value is 0. The syntax is:

BSF *register*,{*register* | *memory*}
 BSR *register*,{*register* | *memory*}

The bit scan instructions work only on 16 bit or 32 bit registers. They cannot be used on memory operands or 8 bit registers. The source register contains the value to be scanned. The destination register should be the register where you want to store the position of the first or last set bit. The BSF (Bit Scan Forward) instruction scans the bits of the source register starting with the 0 bit and working toward the most significant bit. The BSR (Bit Scan Reverse) instruction scans the bits of the source register starting with the most significant bit and working toward the 0 bit.

The following example scans a large bit field. Starting at the beginning of the field, it finds the first nonzero doubleword. Then it finds the first set bit within the doubleword. See Section 20 for more information on the string instructions used in this example.

```
widfield    .DATA
bitfield    EQU      200
            DD      widfield DUP (?)
            .CODE
            .
            .
            .
            cld
            push    ds                ; Load segment of bitfield
            pop     es                ; into ES
            mov     cx,widfield        ; Load maximum count
            xor     eax,eax            ; Set search value to 0
            mov     di,OFFSET bitfield ; Load bitfield address
            repe    scasd              ; Find first nonzero bit
            jecxz   none               ; If none found, get out
            sub     di,4               ; Point back to doubleword
            mov     eax,[di]           ; Else load first nonzero
            bsr     ecx,eax            ; Find first set bit
            .                          ; ECX now contains bit
            .                          ; position
            .                          ; DI points to doubleword
none:        .
```

Shifting and Rotating Bits

The 8086 family processors provide a complete set of instructions for shifting and rotating bits. Bits can be moved right (toward the most significant bits) or left (toward the 0 bit). Values shifted off the end of the operand go into the carry flag.

Shift instructions move bits a specified number of places to the right or left. The last bit in the direction of the shift goes into the carry flag, and the first bit is filled with 0 or with the previous value of the first bit.

Rotate instructions move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate is moved into the first bit position at the other end of the operand. With some variations, the carry bit is used as an additional bit of the operand. Figure 18–1 illustrates the eight variations of shift and rotate instructions for 8 bit operands. Notice that SHL and SAL are exactly the same.

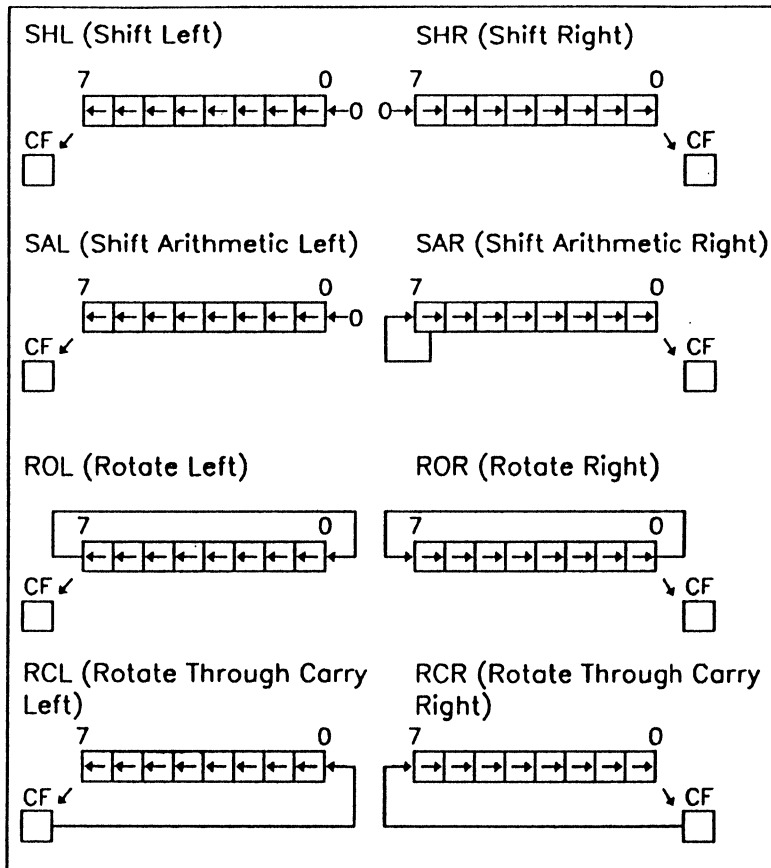


Figure 18-1. Shifts and Rotates

```
SHL {register | memory}, {CL | 1}
SHR {register | memory}, {CL | 1}
SAL {register | memory}, {CL | 1}
SAR {register | memory}, {CL | 1}
ROL {register | memory}, {CL | 1}
ROR {register | memory}, {CL | 1}
RCL {register | memory}, {CL | 1}
RCR {register | memory}, {CL | 1}
```

The format of all the shift instructions is the same. The destination operand should contain the value to be shifted. It will contain the shifted operand after the instruction. The source operand should contain the number of bits to shift or rotate. It can be the immediate value 1 or the CL register. No other value or register is accepted on the 8088 and 8086 processors.

Starting with the 80186 processor, 8 bit immediate values larger than 1 can be given as the source operand for shift or rotate instructions, as shown below:

```
shr    bx,4           ; 9 clocks, 3 bytes on 80286
```

The following statements are equivalent if the program must run the 8088 or 8086:

```
mov    cl,4           ; 2 clocks, 3 bytes on 80286
shr    bx,cl           ; 9 clocks, 2 bytes on 80286
                        ;11 clocks, 5 bytes
```

Multiplying and Dividing by Constants

Shifting right by one has the effect of dividing by two; shifting left by one has the effect of multiplying by two. You can take advantage of this to do fast multiplication and division by common constants. The easiest constants are the powers of two. Shifting left twice multiplies by four, shifting left three times multiplies by eight, and so on.

SHR is used to divide unsigned numbers. SAR can be used to divide signed numbers, but SAR rounds negative numbers down—IDIV always rounds up. Code that divides by using SAR must adjust for this difference. Multiplication by shifting is the same for signed and unsigned numbers, so either SAL or SHL can be used. Both instructions do the same operation.

Since the multiply and divide instructions are the slowest on the 8088 and 8086 processors, using shifts instead can often speed operations by a factor of 10 or more. For example, on the 8088 or 8086 processor, the following statements take 4 clocks:

```
xor    ah,ah        ; Clear AH
shl    ax,1          ; Multiply byte in AL by 2
```

The following statements have the same effect, but take between 74 and 81 clocks on the 8088 or 8086:

```
mov    bl,2          ; Multiply byte in AL by 2
mul    bl
```

The same statements take 15 clocks on the 80286 or between 11 and 16 clocks on the 80386.

Shift instructions can be combined with add or subtract instructions to do multiplication by common constants. These operations are best put in macros so that they can be changed if the constants in a program change. As follows:

```
mul_10  MACRO    factor        ; Factor must be unsigned
        mov     ax,factor      ; Load into AX
        shl     ax,1           ; AX = factor * 2
        mov     bx,ax          ; Save copy in BX
        shl     ax,1           ; AX = factor * 4
        shl     ax,1           ; AX = factor * 8
        add     ax,bx          ; AX = (factor * 8) + (factor * 2)
        ENDM                ; AX = factor * 10

div_u512 MACRO    dividend      ; Dividend must be unsigned
        mov     ax,dividend    ; Load into AX
        shr     ax,1           ; AX = dividend / 2 (unsigned)
        xchg    al,ah          ; xchg is like rotate right 8
                                ; AL = (dividend / 2) / 256
        cbw                    ; Clear upper byte
        ENDM                ; AX = (dividend / 512)
```


Moving Bits to the Least Significant Position

Sometimes a group of bits within an operand needs to be treated as a single unit—for example, to do an arithmetic operation on those bits without affecting other bits. This can be done by masking off the bits, and then shifting them into the least significant positions. After the arithmetic operation is done, the bits are shifted back to the original position and merged with the original bits by using OR. See Section 9 for an example of this operation.

Adjusting Masks

Masks for logical instructions can be shifted to new bit positions. For example, an operand that masks off a bit or group of bits can be shifted to move the mask to a different position.

```
masker      .DATA
DB          00000010b      ; Mask that may change at run time
.CODE
.
.
mov         cl,2            ; Rotate two at a time
mov         bl,57h         ; Load value to be changed
;
rol         masker,cl      ; Rotate two to left      01010111b
or          bl,masker      ; Turn on masked values  00001000b
; New value is 05Fh
rol         masker,cl      ; Rotate two more       01011111b
or          bl,masker      ; Turn on masked values  00100000b
; New value is 07Fh      01111111b
```

This technique is useful only if the mask value is unknown until runtime.

Shifting Multiword Values

Sometimes it is necessary to shift a value that is too large to fit in a register. In this case, you can shift each part separately, passing the shifted bits through the carry flag. The RCR or RCL instructions must be used to move the carry value from the first register to the second. RCR and RCL can also be used to initialize the high or low bit of an operand. Since the carry flag is treated as part of the operand (like using a 9 bit operand), the flag value before the operation is crucial. The carry flag may be set by a previous instruction, or you can set it directly using the CLC (Clear Carry Flag), CMC (Complement Carry Flag), and STC (Set Carry Flag) instructions.

```

mem32      .DATA
            DD      500000
            .CODE
            .
            ; Divide 32 bit unsigned
            ;          by 16
            .
again:      mov     cx,4          ; Shift right 4 500000
            shr     WORD PTR mem32[2],1 ; Shift into carry
            ;          DIV     16
            rcr     WORD PTR mem32[0],1 ; Rotate carry in
            loop    again        ;          31250

```

Shifting Multiple Bits

The 80836 processor has new instructions for shifting multiple bits into an operand. The SHLD (Double Precision Shift Left) instruction shifts a specified group of bits left and into an operand. The SHRD (Double Precision Shift Right) instruction shifts a specified group of bits right and into an operand. The syntax is:

```

SHRD {register | memory},register,{CL | immediate}
SHLD {register | memory},register,{CL | immediate}

```

These instructions take three operands. The first (leftmost) contains the value to be shifted. It must be a 16 bit or 32 bit register or memory operand. The second operand contains the bits to be shifted into the value. It must be a register of the same size as the first operand. The third operand contains the number of bits to shift. It may be an immediate operand or the CL register.

mov	ax,3AF2h	; Load	AX=00111010 11110010	
mov	bx,9C00h	; Load	BX=10011100 00000000	
shld	ax,bx,7	; Shift 7	01111001 0	<-- 7
			10011110	<-- 7
			<hr/>	
			AX=01111001 01001110	
			(794Eh)	

Section 19

Controlling Program Flow

The 8086 family processors provide a variety of ways for controlling the flow of a program.

- Jumping
- Looping
- Setting Bytes Conditionally
- Using Procedures
- Using Interrupts

The four major types of program flow instructions are jumps, loops, procedure calls, and interrupts.

This section tells you how to use these instructions and how to test conditions for the instructions that change program flow conditionally.

Jumping

There are two basic types of jumps in MASM:

- Unconditional Jumps
- Conditional Jumps

Jumps are the most direct method of changing program control from one location to another. At the internal level, jumps work by changing the value of the IP (Instruction Pointer) register from the address of the current instruction to a target address.

Jumps can be short, near, or far. MASM automatically handles near and short jumps, though it may not always generate the most efficient code if the label being jumped to is a forward reference. The size and control of jumps is discussed in Section 11.

Jumping Unconditionally

The JMP instruction is used to jump unconditionally to a specified address. The syntax is:

JMP {*register* | *memory*}

The operand should contain the address to be jumped to. Unlike conditional jumps, whose target address must be short (within 128 bytes), the target address for unconditional jumps can be short, near, or far. See Section 11 for more information on specifying the distance for conditional jumps.

If a conditional jump must be greater than 128 bytes, the construction must be reorganized (except on the 80386). This can be done by reversing the sense of the conditional jump and adding an unconditional jump, as shown in the following example:

```
    cmp     ax,7           ; If AX is 7 and jump is short
    je      close         ; then jump close

    cmp ax,6              ; If AX is 6 and jump is near
    jne close            ; then test opposite and skip
                        ; over
    jmp distant          ; Now jump
    .
    .
close:                  ; Less than 128 bytes from jump
    .
    .
distant:                ; More than 128 bytes from jump
```

An unconditional jump can be used as a form of conditional jump by specifying the address in a register or indirect memory operand. The value of the operand can be calculated at run time, based on user interaction or other factors. You can use indirect memory operands to construct jump tables that work like C switch statements, BASIC ON GOTO statements, or Pascal case statements.

In the next example, an indirect memory operand points to addresses of routines for handling different keystrokes. Notice that the jump table is placed in the code segment. This technique is optional in stand alone assembler programs, but it may be required for procedures called from some languages.

```

        .CODE
        .
        .
        jmp      process          ; Jump over data
ctl_tbl LABEL WORD                ; (required in overlay
                                ; procedures)
        DW      extended         ; Null key (extended code)
        DW      ctrl_a           ; Address of CONTROL-A key
                                ; routine
        DW      ctrl_b           ; Address of CONTROL-B key
                                ; routine
DB char ?

process: push     ds              ;
        push     OFFSET char     ;
        call    ReadKbd         ;
        mov      al,char         ;
        cbw                     ; Convert AL to AX
        mov      bx,ax          ; Copy
        shl     bx,1            ; Convert to address
        jmp      ctl_tbl[bx]    ; Jump to key routine

extended: push     ds            ;
        push     OFFSET char     ;
        call    ReadKbd         ;
        mov      al,char         ;
        .                  ; Use another jump table
        .                  ; for extended keys
ctrl_a:  .                  ; CONTROL-A routine here
        .
        jmp      next

ctrl_b:  .                  ; CONTROL-B routine here
        .
        jmp      next
        .
next:    .                  ; Continue

```

Jumping Conditionally

The most common way of transferring control in assembly language is with conditional jumps. This is a two step process: first test the condition, and then jump if the condition is true or continue if it is false. The syntax is:

Jcondition label

Conditional jump instructions take a single operand containing the address to be jumped to. The distance from the jump instruction to the specified address must be short (less than 128 bytes). If a longer distance is specified, an error will be generated telling the distance of the jump in bytes. See *Jumping Unconditionally*, for information on arranging longer conditional jumps.

Note: *Conditional jumps to forward references are near by default under the 80386 processor. But you can use the SHORT operator to specify short jumps. See Section 11 for information specifying the size of jumps.*

Conditional jump instructions (except JCXZ) use the status of one or more flags as their condition. Thus any statement that sets a flag under specified conditions can be the test statement. The most common test statements use the CMP or TEST instructions. The jump statement can be any one of 31 conditional jump instructions.

Comparing and Jumping

The CMP instruction is specifically designed to test for conditional jumps. It does not change the destination operand, so it can be used to compare two values without changing either of them. Instructions that change operands (such as SUB or AND) can also be used to test conditions.

The CMP instruction compares two operands and sets flags based on the result. It is used to test the following relationships: equal; not equal; greater than; less than; greater than or equal; or less than or equal. The syntax is:

CMP {register | memory},{register | memory | immediate}

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, they cannot both be memory operands.

The jump instructions that can be used with CMP are made up of mnemonic letters combined to indicate the type of jump. The letters are shown below:

Letter	Meaning
J	Jump
G	Greater than (for unsigned comparisons)
L	Less than (for unsigned comparisons)
A	Above (for signed comparisons)
B	Below (for signed comparisons)
E	Equal
N	Not

The mnemonic names always refer to the relationship that the first operand of the CMP instruction has to the second operand of the CMP instruction. For instance, JG tests whether the first operand is greater than the second. Several conditional instructions have two names. You can use whichever name seems more mnemonic in context.

Comparisons and conditional jumps can be thought of as statements in the following format:

IF (value1 relationship value2) THEN GOTO truelabel

Statements of this type can be coded in assembly language by using the following syntax:

```
CMP value1,value2
Jrelationship truelabel
.
.
.
truelabel:
```


Table 19–1 lists conditional jump instructions for each *relationship* and shows the flags that are tested in order to see if *relationship* is true.

Table 19–1. Conditional–Jump Instructions Used after Compare

Jump Condition		Signed Compare	Jump If:	Unsigned Compare	Jump If:
Equal	=	JE	ZF = 1	JE	ZF = 1
Not Equal		JNE	ZF = 1	JNE	ZF = 1
Greater than	>	JG or JNLE	ZF = 0 and SF = OF	JA or JNBE	CF = 0 and ZF = 0
Less than or equal	≤	JLE or JNG	ZF = 1 and SF ≠ OF	JBE or JNA	CF = 1 or ZF = 1
Less than	<	JL or JNGE	SF ≠ OF	JB or JNAE	CF = 1
Greater than or equal	≥	JGE or JNL	SF = OF	JAE or JNB	CF = 0

Internally, the CMP instruction is exactly the same as the SUB instruction, except that the destination operand is not changed. The flags are set according to the result that would have been generated by a subtraction.

```
; If CX is less than -20, then make DX 30, else make DX 20
    cmp    cx,-20      ; If signed CX is smaller than -20
    jl     less        ; Then do stuff at less
    mov     dx,20      ; Else set DX to 20
    jmp     further    ; Finished
less:    mov     dx,30  ; Then set DX to 30
further:
```

The example above shows the basic form of conditional jumps. Notice that in assembly language, if-then-else constructions are usually written in the form if-else-then. This theme has many variations. For example, you may find it more mnemonic to code in the if-then-else format. However, you must then use the opposite jump condition, as shown in the following example:

; If CX is greater than or equal to -20, then make DX 20, else make DX 30

```

        cmp     cx,-20           ; If signed CX is smaller than -20
        jnl     notless         ;   else do stuff at notless
        mov     dx,30           ; Then set DX to 30
        jmp     continue        ; Finished
notless: mov     dx,20           ; Else set DX to 20
continue:

```

The then-if-else format shown in the next example is often more efficient. Do the work for the most likely case, and then compare for the opposite condition. If the condition is true, you are finished.

; DX is 20, unless CX is less than -20, then make DX 30

```

        mov     dx,20           ; DX is 20
        cmp     cx,-20         ; If signed CX is greater than -20
        jge     greatequ       ;   Then done
        mov     dx,30           ; Else set DX to 30
greatequ:

```

This example avoids the unconditional jump used in Examples 1 and 2 and thus is faster even if the less likely condition is true.

Jumping Based on Flag Status

The CMP instruction is the most mnemonic way to set the flags for conditional jumps, but any instruction that changes flags can be used as the test condition. The conditional jump instructions listed below enable you to jump based on the condition of flags rather than on relationships of operands. Some of these instructions have the same effect as instructions listed in Table 19–1.

Instruction	Action
JO	Jumps if the overflow flag is set
JNO	Jumps if the overflow flag is clear
JC	Jumps if the carry flag is set (same as JB)
JNC	Jumps if the carry flag is clear (same as JAE)
JZ	Jumps if the zero flag is set (same as JE)
JNZ	Jumps if the zero flag is clear (same as JNE)
JS	Jumps if the sign flag is set
JNS	Jumps if the sign flag is clear
JP	Jumps if the parity flag is set
JNP	Jumps if the parity flag is clear
JPE	Jumps if parity is even (parity flag set)
JPO	Jumps if parity is odd (parity flag clear)
JCXZ	Jumps if CX is 0

Notice that the JCXZ is the only conditional jump based on the condition of a register (CX) rather than flags. Since JCXZ is usually used with loop instructions, it is discussed in more detail in Setting Bytes Conditionally.

```
        add     ax,bx        ; Add two values
        jo      overflow     ; If value too large, adjust
        .
        .
        .
overflow:                               ; Adjustment routine here
        sub     ax,dx        ; Subtract
        jnz     go_on        ; If the result is not zero,
                               ;   continue
        call    zhandler     ; else do special case
go_on:
```

Testing Bits and Jumping

Like the CMP instruction, the TEST instruction is designed to test for conditional jumps. However, specific bits are compared rather than entire operands. The syntax is:

TEST {*register* | *memory*},{*register* | *memory* | *immediate*}

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, the operands cannot both be memory.

Normally, one of the operands is a mask in which the bits to be tested are the only bits set. The other operand contains the value to be tested. If all the bits set in the mask are clear in the operand being tested, the zero flag will be set. If any of the flags set in the mask are also set in the operand, the zero flag will be cleared.

The TEST instruction is actually the same as the AND instruction, except that neither operand is changed. If the result of the operation is 0, the zero flag is set, but the 0 is not actually written to the destination operand. You can use the JZ and JNZ instructions to jump after the test. JE and JNE are the same and can be used if you find them more mnemonic. As follows:

```

bits      .DATA
          DB      ?
          .CODE
          .
          .
; If bit 2 or bit 4 is set, then call taska

          test    bits,10100b    ; Assume bits is 0D3h      11010011
          jz      go_on          ; If 2 or 4 is set AND    00010100
          call    taska          ; Else continue
          ; Then call taska      00010000
go_on:    ; Jump not taken

          .
          .
          .
          ; If bits 2 and 4 are clear, then call taskb

          test    bits,10100b    ; Assume bits is 0E9h      11101001
          jnz     next           ; If 2 and 4 are clear AND  00010100
          call    taskb          ; Else continue
          ; Then call taskb      00000000
next:    ; Jump not taken

```

Testing and Setting Bits

The 80386 processor has bit test and set instructions. These instructions have two purposes. They can test the status of a bit to control program flow; some of them can also change the value of a specified bit.

```

BT {register | memory},{register | immediate}
BTC {register | memory},{register | immediate}
BTR {register | memory},{register | immediate}
BTS {register | memory},{register | immediate}

```

For each of the instructions, the memory or register destination operand is the target value that will be tested. The register or immediate source operand specifies the number of the bit to be tested in the destination operand. The four bit-testing instructions are described below:

BT the Bit Test instruction examines the specified bit in the target value and puts a copy in the carry flag. The carry flag can then be used by another instruction such as a conditional jump. For example, assume BX points to a bit field and CX contains 4 in the following statements:

```
bt    [bx],cx    ; Put bit 4 of bit field
                ; pointed to by BX in carry
jc    somewhere  ; Jump if carry set
```

The same thing could be done less efficiently on other 8086 family processors with the following statements:

```
mov    ax,[bx]    ; Load value pointed to by BX
shr    ax,cl      ; Shift bit 4 to first position
test   ax,1       ; See if bit is set
jnz    somewhere  ; Jump if it is
```

This instruction is only useful if the source operand is not known until run time. If the source operand is a constant, the **TEST** instruction (see Testing Bits and Jumping) is more efficient.

BTC the Bit Test and Complement instruction examines the specified bit in the target value and puts a copy in the carry flag. It then reverses the value of the bit. For example, assume BX points to a bit field and CX contains 4 in the following statements:

```
btc    [bx],cx    ; Put bit 4 of bit field in carry
                ; and toggle bit 4
jc     somewhere  ; Jump if carry set
```

BTR the Bit Test and Reset instruction examines the specified bit in the target value and puts a copy in the carry flag. It then clears the bit. For example, assume BX points to a bit field and CX contains 4 in the following statements:

```
btr    [bx],cx    ; Put bit 4 of bit field in carry
                ; and clear bit 4
jc     somewhere  ; Jump if carry set
```

BTS the Bit Test and Set instruction examines the specified bit in the target value and puts a copy in the carry flag. It then sets the bit. For example, assume BX points to a bit field and CX contains 4 in the following statements:

```
bts    [bx],cx    ; Put bit 4 of bit field in carry
                ; and set bit 4
jc     somewhere  ; Jump if carry was set
```

In the following example, a bit field made up of error flags is tested. If the bit flag being tested is set, indicating an error, the flag is turned off and control is directed to a label where the error is corrected.

```
flag    .DATA
error   RECORD  a:3=0,b:2=0,c:1=0,d:2=0,e:1=0,f:1=0
        flag    < >
        .CODE
        .
        .
        .
        btr     error,c
        jc      fixc
        .
        .
fixa:    .
```

Looping

The 8086 family of processors has several instructions specifically designed for creating loops of repeated instructions. In addition, you can create loops using conditional jumps.

LOOP label
LOOPE label
LOOPZ label
LOOPNE label
LOOPNZ label
JCXZ label

The LOOP instruction is used for loops with a set number of iterations. For example, it can be used in constructions similar to the for loops of BASIC, C, and Pascal, and the do loops of FORTRAN.

A single operand specifies the address to jump to each time through the loop. The CX register is used as a counter for the number of times to loop. On each iteration, CX is decremented. When CX reaches 0, control passes to the instruction after the loop.

The LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions are used in loops that check for a condition. For example, they can be used in constructions similar to the while loops of BASIC, C, and Pascal; the repeat loops of Pascal; and the do loops of C.

The LOOPE (also called LOOPZ) instruction can be thought of as meaning loop while equal. Similarly, LOOPNE (also called LOOPNZ) instruction can be thought of as meaning loop while not equal. A single short memory operand specifies the address to loop to each time through. The CX register can specify a maximum number of times to go through the loop. The CX register can be set to a number that is out of range if you do not want a maximum count.

The JCXZ instruction (and its 32 bit 80386 extension, JECXZ) are often used in loop structures. For example, it may be used in loops that check a condition at the start of the loop rather than at the end. Unlike the loop instruction, JCXZ does not decrement CX, so the programmer must use another statement to decrement the count.

Note: *Unlike conditional jump instructions, which can jump to either a near or a short label under the 80386, the loop instructions, JCXZ instruction, and JECXZ instruction always jump to a short label.*

The following example uses the loop instruction to execute a task 200 times:

```
; For 0 to 200 do task
next:    mov     cx,200        ; Set counter
        .
        .
        .
        loop    next         ; Do again
                                ; Continue after loop
```

This loop has the same effect as the following statements:

```
; For 0 to 200, do task
next:    mov     cx,200        ; Set counter
        .
        .
        .
        dec     cx
        cmp     cx,0
        jne     next         ; Do again
                                ; Continue after loop
```


The first version is more efficient as well as easier to understand. However, there are situations in which you must use conditional jump instructions rather than loop instructions. For example, conditional jumps are often required for loops that test several conditions.

If the counter in CX is variable because of previous instructions, you should use the JCXZ instruction to check for 0, as shown in Example 2. Otherwise, if CX is 0, it will be decremented to -1 in the first iteration and will continue through 65,535 iterations before it reaches 0 again. For example:

```
; For 0 to CX do task

next:      jcxz   done          ; CX counter set
          .           ; previously
          .           ; Check for 0
          .           ; Do the task here
          loop   next          ; Do again
done:      .           ; Continue after loop
          .           ; While AX is not 128, do task
          mov    cx,0FFFFh     ; Set count too high to
          .           ; interfere
wend:     .           ; Do the task here
          .
          .
          cmp    ax,128        ; Is it 128?
          loopne wend          ; No? Repeat
          .                   ; Yes? Continue
```

Setting Bytes Conditionally

The 80386 processor has a new group of instructions for setting bytes conditionally. These instructions test the condition of specified flags, and depending on the result, set a memory operand either to 1 or to 0. They can be used to set byte variables that are used as Boolean flags. The syntax is:

SET*condition* {*register* | *memory*}

Conditional set instructions test conditions in the same way as conditional jump instructions, except that instead of jumping if the condition is met, they set a specified byte. For example, SETZ is similar to JZ, SETNE is similar to JNE, and so on. See Jumping Unconditionally for more information on how flags are tested for conditional jumps.

Conditional set instructions require one 8 bit operand, which can be either a register or a memory operand. If the condition tested by the instruction is true, the operand is set to 1. Otherwise the operand is set to 0.

Conditional set instructions are usually preceded by a `CMP` or `TEST` instruction, although any instruction that sets flags can be used to test for the condition. As follows:

```
.DATA
bigflag    DB    ?           ; Boolean flag
amount     DW    ?           ; Size variable to be set at run
                                   ; time

.CODE
.
.           ; Size is set
.
; bigflag = amount > 1000

    cmp     size,1000 ; Is size greater than 1000?
    setg    bigflag   ; If greater, bigflag = 1
                                   ; else bigflag = 0
```

In the example, the Boolean variable `bigflag` is set according to a comparison of two other values. Some languages (such as BASIC) set the result of true relational statements to `-1` rather than `1`. To make the code compatible with such compilers, you should negate the value after setting it. For example, add the following line to the previous example:

```
neg    bigflag   ; Negate result
```

This statement would be necessary for BASIC, since the expression `BIGFLAG=SIZE>1000` evaluates to `-1`. It would not be necessary for C, since the expression `bigflag = size > 1000` evaluates to `1`.

Using Procedures

This subsection provides the following information on the use of procedures:

- Calling Procedures
- Defining Procedures
- Passing Arguments on the Stack
- Using Local Variables
- Setting Up Stack Frames

Procedures are units of code that do a specific task. They provide a way of modularizing code so that a task can be accomplished from any point in a program without using the same code in each place. Assembly language procedures are comparable to functions in C; subprograms, functions, and subroutines in BASIC; procedures and functions in Pascal; or routines and functions in FORTRAN.

Two instructions and two directives are usually used in combination to define and use assembly language procedures. The CALL instruction is used to call procedures defined elsewhere. The RET instruction is used to return control from a called procedure to the code that called it. The PROC and ENDP directives normally mark the beginning and end of a procedure definition, as described in Defining Procedures.

The CALL and RET instructions use the stack to keep track of the location of the procedure. The CALL instruction pushes the calling address onto the stack and then jumps to the starting address of the procedure.

The RET instruction pops the address pushed by the CALL instruction and returns control to the instruction following the call.

Every CALL must have a RET to restore the stack to its status before the CALL. Calls may be nested.

Calling Procedures

The CALL instruction saves the address following the instruction on the stack and passes control to a specified address. The syntax is:

`CALL {register | memory}`

The address is usually specified as a direct memory operand. However, the operand can also be a register or indirect memory operand containing a value calculated at run time. This enables you to write call tables similar to the jump table illustrated in Comparing and Jumping.

Calls can be near or far. Near calls push only the offset portion of the calling address. Far calls push both the segment and offset. You must give the type of far calls to forward referenced labels using the FAR type specifier and the PTR operator. For example, use the following statement to make a far call to a label that has not been earlier defined or declared external in the source code:

```
call FAR PTR task
```

Defining Procedures

Procedures are defined by labeling the start of the procedure and placing a RET instruction at the end. There are several variations on this syntax.

```
label PROC [[NEAR | FAR]]  
statements  
RET [[constant]]  
label ENDP
```

Procedures are normally defined by using the PROC directive at the start of the procedure and the ENDP directive at the end. The RET instruction is normally placed immediately before the ENDP directive. The size of the RET instruction automatically matches the size defined by the PROC directive.

```
label:  
statements  
RETN [[constant]]  
  
label LABEL FAR  
statements  
RETF [[constant]]
```

The RET instruction can be extended to RETN (Return Near) or RETF (Return Far) to override the default size. This enables you to define and use procedures without the PROC and ENDP directives, as shown in Syntax 2 and Syntax 3 above. However, with this method, the programmer is responsible for making sure the size of the CALL matches the size of the RET.

The RET instruction (and its RETF and RETN variations) allows a constant operand that specifies a number of bytes to be added to the value of the SP register after the return. This operand can be used to adjust for arguments passed to the procedure before the call, as shown in the example in Using Local Variables.

The following example shows the recommended way of making calls with MASM.

```
call task      ; Call is near because
               ; procedure is near
               ; Return comes to here
.
.
task PROC NEAR ; Define task to be near
.
               ; Instructions of task go here
.
ret            ; Return to instruction after
               ; call
task ENDP      ; End task definition
```

The example below shows another method that programmers who are used to other assemblers may find more familiar.

```
call NEAR PTR task ; Call is declared near
.                  ; Return comes to here
.
task:              ; Procedure begins with near
                  ; label
.
                  ; Instructions go here.
retn              ; Return declared near
```

This method gives more direct control over procedures, but the programmer must make sure that calls have the same size as corresponding returns.

For example, if a call is made with the statement

```
call NEAR PTR task
```

the assembler does a near call. This means that one word (the offset following the calling address) is pushed onto the stack. If the return is made with the statement

```
retf
```

two words are popped off the stack. The first will be the offset, but the second will be whatever happened to be on the stack before the call. Not only will the popped value be meaningless, but the stack status will be incorrect, causing the program to fail.

Passing Arguments on the Stack

Procedure arguments can be passed in various ways. For example, values can be passed to a procedure in registers or in variables. However, the most common method of passing arguments is to use the stack.

The arguments are pushed onto the stack before the call. After the call, the procedure retrieves and processes them. At the end of the procedure, the stack is adjusted to account for the arguments.

In some languages, pointers to the arguments are passed to the procedure; in others the arguments themselves are passed. The order in which arguments are passed (whether the first argument is pushed first or last) also varies according to the language. Finally, in some languages, the stack is adjusted by the RET instruction in the called procedure; in others the code immediately following the CALL instruction adjusts the stack.

The following example shows one method of passing arguments to a procedure. This method is similar to the way procedures are called in C.

; C style procedure call and definition

```

        mov     ax,10      ; Load and
        push    ax         ; push constant as third argument
        push    arg2       ; Push memory as second argument
        push    cx         ; Push register as first argument
        call    addup       ; Call the procedure
        add     sp,6        ; Destroy the pushed argument
        .              ; (equivalent to three pops)
        .
addup    PROC    NEAR      ; Return address for near call
                                ; takes two bytes
        push    bp         ; Save base pointer takes two bytes
                                ; so arguments start at 4th byte
        mov     bp,sp       ; Load stack into base pointer
        mov     ax,[bp+4]   ; Get first argument from
                                ; 4th byte above pointer
        add     ax,[bp+6]   ; Add second argument from
                                ; 6th byte above pointer
        add     ax,[bp+8]   ; Add third argument from
                                ; 8th byte above pointer
        pop     bp         ; Restore BP
        ret                     ; Return result in AX
addup    ENDP

```

Figure 19–1 shows the stack condition at key points in the process.

Note: *Arguments passed on the stack in assembler routines cannot be accessed by name with a symbolic debugger. They can be accessed by an expression that specifies their stack position.*

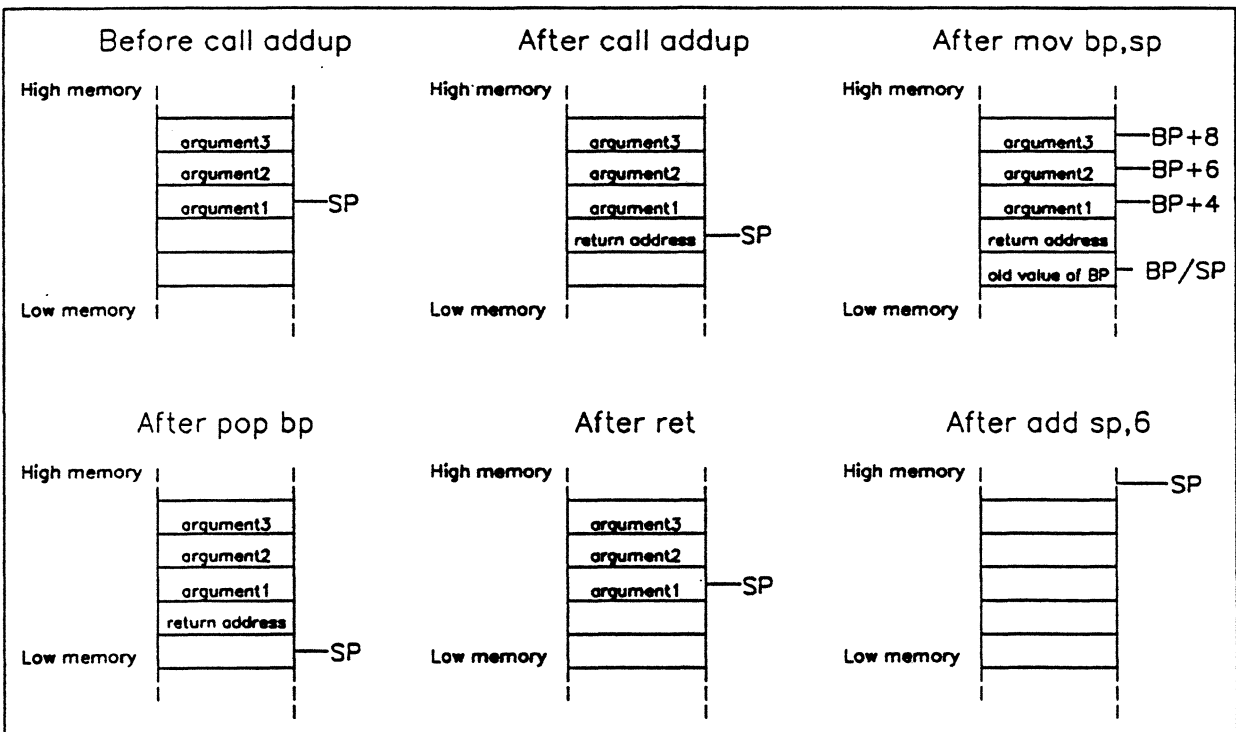


Figure 19-1. Procedure Arguments on the Stack

Using Local Variables

In high level languages, local variables are variables known only within a procedure. These variables are usually stored on the stack. Assembly language programs can use the same concept. These variables should not be confused with labels or variable names that are local to a module, as described in Section 10.

Local variables are created by saving stack space for the variable at the start of the procedure. The variable can then be accessed by its position in the stack. At the end of the procedure, the stack pointer is restored to restore the memory used by local variables.

In the following example, two bytes are subtracted from the SP register to make room for a local word variable. This variable can then be accessed as [bp-2].

```

                push    ax            ; Push one argument
                call    task          ; Call
                .
                .
                .
arg            EQU      < [bp+4] >    ; Name for argument
loc            EQU      < [bp-2] >    ; Name for local variable
task          PROC     NEAR
push          bp                    ; Save base pointer
                mov     bp,sp         ; Load stack into base pointer
                sub     sp,2          ; Save two bytes for local
                                     ; variable
                .
                .
                .
                mov     loc,3         ; Initialize local variable
                add     ax,loc         ; Add local variable to AX
                sub     arg,ax         ; Subtract local from argument
                .                    ; Use loc and arg in other
                                     ; operations
                .
                .
                mov     sp,bp         ; Adjust for stack variable
                pop     bp            ; Restore base
                ret     2              ; Return result in AX and pop
                                     ; two bytes to adjust stack
task          ENDP

```


In the example, this value is given the name `loc` with a text equate. Notice that the instruction `mov sp,bp` is given at the end to restore the original value of `SP`. The statement is only required if the value of `SP` is changed inside the procedure (usually by allocating local variables).

The argument passed to the procedure is returned with the `RET` instruction. Contrast this to the example in *Passing Arguments on the Stack* in which the calling code adjusts for the argument. Figure 19–2 shows the state of the stack at key points in the process.

Note: *Local variables created in assembler routines cannot be accessed by name with a symbolic debugger. They can be accessed by an expression that specifies their stack position.*

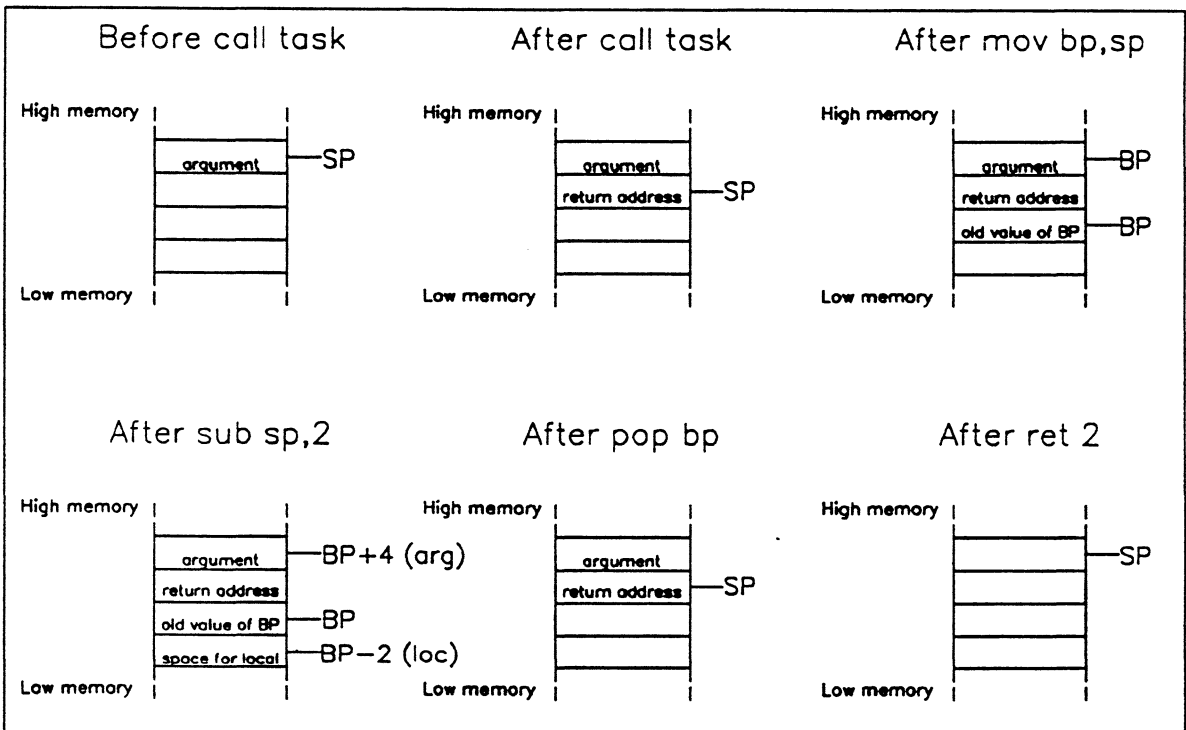


Figure 19-2. Local Variables on the Stack

Setting Up Stack Frames

Starting with the 80186 processor, the ENTER and LEAVE instructions are provided for setting up a stack frame. These instructions do the same thing as the multiple instructions at the start and end of procedures in the Microsoft calling conventions. The syntax is:

```
ENTER framesize, nestinglevel
statements
LEAVE
```

The ENTER instruction takes two constant operands. *framesize* (a 16 bit constant) specifies how many bytes to reserve for local variables. *nestinglevel* (an 8 bit constant) specifies the level at which the procedure is nested. This operand should always be 0 when writing procedures for BASIC, C, and FORTRAN. *nestinglevel* can be greater than 0 with Pascal and other languages that enable procedures to access the local variables of calling procedures.

The LEAVE instruction reverses the effect of the last ENTER instruction by restoring BP and SP to their values before the procedure call as in the following example:

```
task      PROC      NEAR
          enter      6,0      ; Set stack frame and reserve 6
          .           ; bytes for local variables
          .           ; Do task here
          .
          leave      ; Restore stack frame
          ret        ; Return
task      ENDP
```

The preceding example has the same effect as the code in the following example.

```
task      PROC      NEAR
          push      bp      ; Save base pointer
          mov       bp,sp    ; Load stack into base pointer
          sub       sp,6     ; Reserve 6 bytes for local
          .             ; variables
          .             ; Do task here
          .
          mov       sp,bp    ; Restore stack pointer
          pop       bp      ; Restore base
          ret       ; Return
task      ENDP
```

The code in the first example, above takes fewer bytes, but is slightly slower.

Using Interrupts

Interrupts are a special form of routines that are called by number instead of by address. They can be initiated by hardware devices as well as by software. Hardware interrupts are called automatically whenever certain events occur in the hardware.

Interrupts can have any number from 0 to 255. Most of the interrupts with lower numbers are reserved for use by the processor or the operating system.

The programmer can call existing interrupts with the INT instruction. Interrupt routines can also be defined or redefined to be called later. For example, an interrupt routine that is called automatically by a hardware device can be redefined so that its action is different. Two interrupt routines that are sometimes used by applications programmers are listed below:

Interrupt	Description
0	Divide overflow. Called automatically when the quotient of a divide operation is too large for the source operand or when a divide by zero is attempted.
4	Overflow. Called by the INTO instruction if the overflow flag is set.

Calling Interrupts

Interrupts are called with the INT instruction.

INT *interruptnumber*

INTO

The INT instruction takes an immediate operand with a value between 0 and 255.

Registers may be used to pass arguments to functions. Some interrupts and functions return values in certain registers. Register use varies for each interrupt.

When the instruction is called, the processor takes the following six steps:

- Looks up the address of the interrupt routine in the interrupt descriptor table. In real mode, this table starts at the lowest point in memory (segment 0, offset 0) and consists of four bytes (two segment and two offset) for each interrupt. Thus the address of an interrupt routine can be found by multiplying the number of the interrupt by four.
- Pushes the flags register, the current code segment (CS), and the current instruction pointer (IP).
- Clears the trap (TF) and interrupt enable (IF) flags.
- Jumps to the address of the interrupt routine, as specified in the interrupt description table.
- Executes the code of the interrupt routine until it encounters an IRET instruction.
- Pops the instruction pointer, code segment, and flags.

Figure 19–3 illustrates how interrupts work.

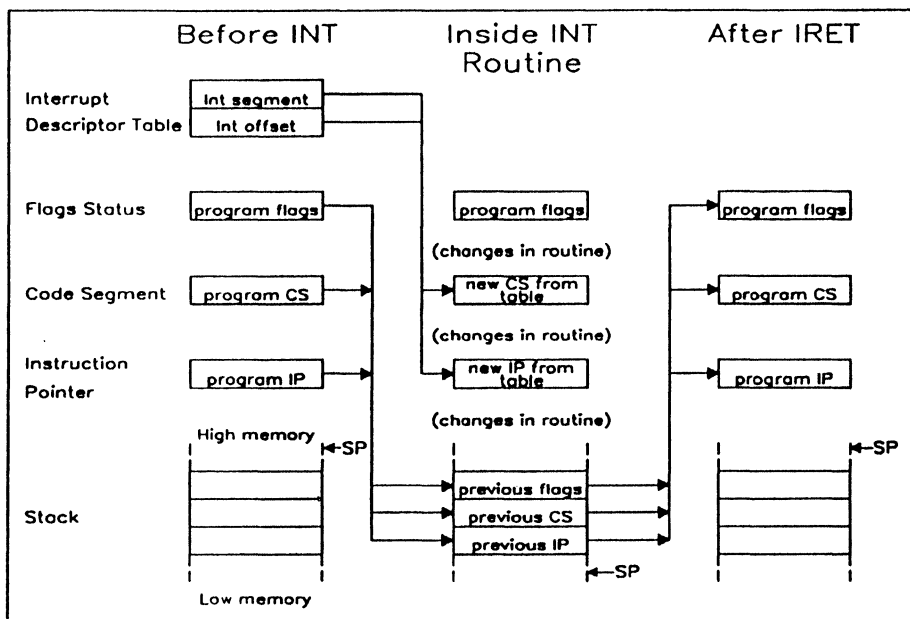


Figure 19-3. Operation of Interrupts

The INTO (Interrupt on Overflow) instruction is a variation of the INT instruction. It calls interrupt 04h if called when the overflow flag is set. By default, the routine for interrupt 4 simply consists of an IRET so that it returns without doing anything. However, you can write your own overflow interrupt routine. Using INTO is an alternative to using JO (Jump on Overflow) to jump to an overflow routine. Defining and Redefining Interrupt Routines gives an example of this.

The CLI (Clear Interrupt Flag) and STI (Set Interrupt Flag) instructions can be used to turn interrupts on or off. You can use CLI to turn interrupt processing off so that an important routine cannot be stopped by a hardware interrupt. After the routine has finished, use STI to turn interrupt processing back on. Interrupts received while interrupt processing was turned off by CLI are saved and executed when STI turns interrupts back on.

For more information on interrupts, refer to the CTOS Open Programming Practices and Standards.

Section 20

Processing Strings

The 8086 family processors have a full set of instructions for manipulating strings:

- Setting Up String Operations
- Moving Strings
- Searching Strings
- Comparing Strings
- Filling Strings
- Loading Values from Strings
- Transferring Strings to and from Ports

In the discussion of these instructions, the term string refers not only to the common definition of a string—a sequence of bytes containing characters—but to any sequence of bytes or words (or doublewords on the 80386).

The following instructions are provided for 8086 family string functions:

Instruction	Description
--------------------	--------------------

MOVS	Moves string from one location to another
SCAS	Scans string for specified values
CMPS	Compares values in one string with values in another
LODS	Loads values from a string to accumulator register
STOS	Stores values from accumulator register to a string
INS	Transfers values from a port to memory
OUTS	Transfers values from memory to a port

All these instructions use registers in the same way and have a similar syntax. Most are used with the repeat instruction prefixes: REP, REPE, REPNE, REPZ, and REPNZ.

This chapter first explains the general format for string instructions and then tells you how to use each instruction.

Setting Up String Operations

The string instructions all work in a similar way. Once you understand the general procedure, it is easy to adapt the format for a particular string operation. The five steps are listed below:

- Make sure the direction flag indicates the direction in which you want the string to be processed. If the direction flag is clear, the string will be processed up (from low addresses to high addresses). If the direction flag is set, the string will be processed down (from high addresses to low addresses). The CLD instruction clears the flag, while STD sets it.
- Load the number of iterations for the string instruction into the CX register. For instance, if you want to process a 100 byte string, load 100. If a string instruction will be terminated conditionally, load the maximum number of iterations that can be done without an error.
- Load the starting offset address of the source string into DS:SI and the starting address of the destination string into ES:DI. Some string instructions take only a destination or source (shown in Table 20–1). Normally the segment address of the source string should be DS, but you can use a segment override with the string instruction to specify a different segment. You cannot override the segment address for the destination string. Therefore you may need to change the value of ES.
- Choose the appropriate repeat prefix instruction. Table 20–1 shows the repeat prefixes that can be used with each instruction.
- Put the appropriate string instruction immediately after the repeat prefix (on the same line).

String instructions have two basic forms, as shown below:

```
[ repeatprefix ] stringinstruction [ ES: [ destination, ] ] [ segmentregister: source ]
```

The string instruction can be given with the source and/or destination as operands. The size of the operand or operands indicates the size of the objects to be processed by the string. Note that the operands only specify the size. The actual values to be worked on are the ones pointed to by DS:SI and/or ES:DI. No error is generated if the operand is not the same as the actual source or destination. One important advantage of this syntax is that the source operand can have a segment override. The destination operand is always relative to ES and cannot be overridden.

```
[[ repeatprefix ]] stringinstructionB
[[ repeatprefix ]] stringinstructionW
[[ repeatprefix ]] stringinstructionD (80386 only)
```

The letter B or W appended to the stringinstruction indicates bytes or words; the letter D indicates doublewords on the 80386. With a letter appended to a string instruction, no operand is allowed. For instance, MOVSB can be given with byte operands to move bytes or with word operands to move words. As an alternative, MOVSB can be given with no operands to move bytes or MOVSW can be given with no operands to move words.

Instructions that specify the size in the name never accept operands. Therefore, the following statement is illegal:

```
lodsb es:0 ; Illegal – no operand
           ; allowed
```

Instead, the statement must be coded as shown below:

```
lodsb BYTE PTR es:0 ; Legal – use type
                    ; specifier
```

If a repeat prefix is used, it can be one of the following instructions:

Instruction	Description
REP	Repeats for a specified number of iterations. The number is given in CX.
REPE or REPZ	Repeats while equal. The maximum number of iterations should be specified in CX.
REPNE or REPNZ	Repeats while not equal. The maximum number of iterations should be specified in CX.

REPE is the same as REPZ, and REPNE is the same as REPNZ. You can use whichever name you find more mnemonic. The prefixes ending with E are used in syntax listings and tables in the rest of this chapter.

Table 20–1 lists each string instruction with the type of repeat prefix it uses and whether the instruction works on a source, a destination, or both.

Table 20–1. Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	REP	Both	DS:SI, ES:DI
SCAS	REPE/REPNE	Destination	ES:DI
CMPS	REPE/REPNE	Both	ES:DI, DS:SI
LODS	None	Source	DS:SI
STOS	REP	Destination	ES:DI
INS	REP	Destination	ES:DI
OUTS	REP	Source	DS:SI

At run time, a string instruction preceded by a repeat sequence causes the processor to take the following steps:

- Checks the CX registers and exits from the string instruction if CX is 0.
- Performs the string operation once.
- Increases SI and/or DI if the direction flag is cleared. Decreases SI and/or DI if the direction flag is set. The amount of increase or decrease is one for byte operations, two for word operations, or four for doubleword operations (80386 only).
- Decrements CX (no flags are modified).
- If the string instruction is SCAS or CMPS, checks the zero flag and exits if the repeat condition is false—that is, if the flag is set with REPE or REPZ or if it is clear with REPNE or REPNZ.
- Goes to the next iteration (step 1).

Although string instructions (except LODS) are most often used with repeat prefixes, they can also be used by themselves. In this case, the SI and/or DI registers are adjusted as specified by the direction flag and the size of operands. However, you must decrement the CX register and set up a loop for the repeated action.

Note: *Although you can use a segment override on the source operand, a segment override combined with a repeat prefix can cause problems in certain situations on all processors except the 80386. If an interrupt occurs during the string operation, the segment override is lost and the rest of the string operation processes incorrectly. Segment overrides can be used safely when interrupts are turned off, when a string instruction is used without a segment override, or when a 80386 processor is used.*

Moving Strings

The MOVS instruction is used to move data from one area of memory to another.

[REP] MOVS [ES:]*destination*,[*segmentregister*:]*source*

[REP] MOVSB

[REP] MOVSW

[REP] MOVSD (80386 only)

To move the data, load the count and the source and destination addresses into the appropriate registers, as discussed in Setting Up String Operations. Then use the REP instruction with the MOVS instruction.

The following example shows how to move a string by using string instructions.

```
source      .MODEL    small
destin      .DATA
            DB        10 DUP ('0123456789')
            DB        100 DUP (?)
            .CODE
            mov        ax,@data          ; Load same segment
            mov        ds,ax             ; to both DS
            mov        es,ax             ; and ES
            .
            .
            cld                          ; Work upward
            mov        cx,100             ; Set iteration count to
                                         ; 100
            mov        si,OFFSET source   ; Load address of source
            mov        di,OFFSET destin   ; Load address of
                                         ; destination
            rep        movsb              ; Move 100 bytesFor
```

comparison, the example below shows a much less efficient way of doing the same operation without string instructions.

```
source      .MODEL    small
destin      .DATA
            DB        10 DUP ('0123456789')
            DB        100 DUP (?)
            .CODE
            .
            .                            ; Assume ES = DS
            .
            mov        cx,100             ; Set iteration count to
                                         ; 100
            mov        si,OFFSET source   ; Load offset of source
            mov        di,OFFSET destin   ; Load offset of
                                         ; destination
repeat:      mov        al,es:[si]         ; Get a byte from source
            mov        [di],al            ; Put it in destination
            inc        si                  ; Increment source pointer
            inc        di                  ; Increment destination
                                         ; pointer
            loop       repeat              ; Do it again
```

Both examples illustrate how to move byte strings in a small model program in which DS already points to the segment containing the variables. In such programs, ES can be set to the same value as DS.

There are several variations on this. If the source string was not in the current data segment, you could load the starting address of its segment into ES. Another option would be to use the MOVS instruction with operands and give a segment override on the source operand. For example, you could use the following statement if ES pointed to both the source and the destination strings:

```
rep movs destin,es:source
```

It is sometimes faster to move a string of bytes as words (or as doublewords on the 80386). You must adjust for any odd bytes, as shown in the following example. Assume the source and destination are already loaded.

```

mov     cx,count           ; Load count
shr     cx,1              ; Divide by 2 (carry will
                          ; be set
                          ; if count is odd)
rep     movsw             ; Move words
rcr     cx,1              ; If odd, make CX 1
rep     movsb             ; Move odd byte if there
                          ; is one

```

Searching Strings

The SCAS instruction is used to scan a string for a specified value. The syntax is:

```

[[ REPE | REPNE ] SCAS [[ ES: ]destination
[[ REPE | REPNE ] SCASB
[[ REPE | REPNE ] SCASW
[[ REPE | REPNE ] SCASD (80386 only)

```

SCAS and its variations work only on a destination string, which must be pointed to by ES:DI. The value to scan for must be in the accumulator register—AL for bytes, AX for words, or EAX (80386 only) for doublewords.

The SCAS instruction works by comparing the value pointed to by DI with the value in the accumulator. If the values are the same, the zero flag is set. Thus the instruction only makes sense when used with one of the repeat prefixes that checks the zero flag.

If you want to search for the first occurrence of a specified value, use the REPNE or REPNZ instruction. If the value is found, ES:DI will point to the value immediately after the first occurrence. You can decrement DI to make it point to the first matching value.

If you want to search for the first value that does not have a specified value, use REPE or REPZ. If the value is found, ES:DI will point to the position after the first nonmatching value. You can decrement DI to make it point to the first nonmatching value.

After a REPNE SCAS, the zero flag will be cleared if no match was found. After a REPE SCAS, the zero flag will be set if no nonmatch was found.

```
.DATA
string    DB      "The quick brown fox jumps over the lazy dog"
lstring   EQU     $-string          ; Length of string
pstring   DD      string            ; Far pointer to string
.CODE
.
.
.
cld                          ; Work upward
mov        cx,lstring        ; Load length of string
les        di,pstring        ; Load address of string
mov        al,'z'            ; Load character to find
repne     scasb              ; Search
jnz        notfound          ; CX is 0 if not found
.                            ; ES:DI points to
.                            ;   character
.                            ;   after first 'z'
.
notfound:                      ; Special case for not
                               ;   found
```

This example assumes that ES is not the same as DS, but that the address of the string is stored in a pointer variable. The LES instruction is used to load the far address of the string into ES:DI.

Comparing Strings

The CMPS instruction is used to compare two strings and point to the address where a match or nonmatch occurs.

```
[[ REPE | REPNE ]] CMPS [[ segment register: ]source, [[ ES: ]destination
[[ REPE | REPNE ]] CMPSB
[[ REPE | REPNE ]] CMPSW
[[ REPE | REPNE ]] CMPSD (80386 only)
```

The count and the addresses of the strings are loaded into registers, as described in Setting Up String Operations. Either string can be considered the destination or source string unless a segment override is used. Notice that unlike other instructions, CMPS requires that the source be on the left.

The CMPS instruction works by comparing in turn each value pointed to by DI with the value pointed to by SI. If the values are the same, the zero flag is set. Thus the instruction makes sense only when used with one of the repeat prefixes that checks the zero flag.

If you want to search for the first match between the strings, use the REPNE or REPNZ instruction. If a match is found, ES:DI and DS:SI will point to the position after the first match in the respective strings. You can decrement DI or SI to point to the match.

If you want to search for a nonmatch, use REPE or REPZ. If a nonmatch is found, ES:DI and DS:SI will point to the position after the first nonmatch in the respective strings. You can decrement DI or SI to point to the nonmatch.

After a REPNE CMPS, the zero flag will be cleared if no match was found. After a REPE CMPS, the zero flag will be set if no nonmatch was found.

The following example assumes that the strings are in different segments. Both segments must be initialized to the appropriate segment register.

```
.MODEL    large
.DATA
string1   DB      The quick brown fox jumps over the lazy dog
.FARDATA
string2   DB      "The quick brown dog jumps over the lazy fox"
lstring   EQU     $-string2
.CODE
mov       ax,@data           ; Load data segment
mov       ds,ax              ; into DS
mov       ax,@fardata        ; Load far data segment
mov       es,ax              ; into ES
.
.
cld                               ; Work upward
mov       cx,lstring          ; Load length of string
mov       si,OFFSET string1   ; Load offset of string1
mov       di,OFFSET string2   ; Load offset of string2
repe      cmpsb               ; Compare
jnz       allmatch            ; CX is 0 if no nonmatch
dec       si                  ; Adjust to point to
                                ; nonmatch
dec       di                  ; in each string
.
.
allmatch: .                    ; Special case for all
                                ; match
```

Filling Strings

The STOS instruction is used to store a specified value in each position of a string.

```
[ REP ] STOS [ ES: ]destination
[ REP ] STOSB
[ REP ] STOSW
[ REP ] STOSD (80386 only)
```

The string is considered the destination, so it must be pointed to by ES:DI. The length and address of the string must be loaded into registers, as described in Setting Up String Operations.

The value to store must be in the accumulator register—for bytes, AX for words, or EAX (80386 only) for doublewords. For each iteration specified by the REP instruction prefix, the value in the accumulator is loaded into the string.

The following example loads 100 bytes containing the character a. Notice that this is done by storing 50 words rather than 100 bytes. This makes the code faster by reducing the number of iterations. You would have to adjust for the last byte if you wanted to fill an odd number of bytes.

```

                .MODEL    small
                .DATA
destin          DB        100 DUP ?
                .CODE
                .          ; Assume ES = DS
                .
                .
                cld        ; Work upward
                mov        ax,'aa'      ; Load character to fill
                mov        cx,50        ; Load length of string
                mov        di,OFFSET destin ; Load address of
                                      ; destination
                rep        stosw        ; Store 'a' into array

```

Loading Values from Strings

The LODS instruction is used to load a value from a string into a register.

LODS [*segmentregister*:]*source*

LODSB

LODSW

LODSD (80386 only)

The string is considered the source, so it must be pointed to by DS:SI. The value is always loaded from the string into the accumulator register—AL for bytes, AX for words, or EAX (80386 only) for doublewords. Unlike other string instructions, LODS is not normally used with a repeat prefix since there is no reason to move a value repeatedly to a register. However, LODS does adjust the DI register as specified by the direction flag and the size of operands. The programmer must code the instructions to use the value after it is loaded.

Processing Strings

The following example loads, processes, and displays each byte in a string of bytes.

```
stuff      .DATA
           DB      0,1,2,3,4,5,6,7,8,9
           .CODE
           .
           .
           cld                      ; Work upward
           mov      cx,10            ; Load length
           mov      si,OFFSET stuff ; Load offset of source
           xor      ah,ah           ; clear ah
get:        lodsb                    ; Get a character
           add      al,48            ; Convert to ASCII
           push     ax              ;
           call     PutChar         ;
           loop     get             ; Repeat
```

In the following example, both LODSB and STOSB are used without repeat prefixes.

```
buffer     .DATA
           DB      80 DUP(?)        ; Create buffer for
                                     ; argument string
           .CODE
start:      mov      ax,@data        ; Initialize DS
           mov      ds,ax
           cld                      ; Work upward
           mov      cl,BYTE PTR es:[80h] ; Load length of arguments
           xor      ch,ch
           mov      di,OFFSET buffer ; Load offset of buffer
           mov      si,82h           ; Load position of
                                     ; argument string
           mov      dx,es            ; Exchange ES and DS
           mov      ax,ds
           mov      es,ax
           mov      ds,dx
another:    lodsb                    ; Get a character
           cmp      al,'a'           ; Is it high enough to be
                                     ; upper?
           jb       noway            ; No? Check
           cmp      al,'z'           ; Is it low enough to be
                                     ; letter?
           ja       noway            ; Yes? Convert to
           sub      al,32             ; uppercase
noway:      stosb
           loop     another          ; Repeat
           mov      dx,es            ; Restore ES and DS
           mov      ax,ds
           mov      es,ax
           mov      ds,dx
```

Transferring Strings to and from Ports

The INS instruction reads a string from a port to memory, and the OUTS instruction writes a string from memory to a port.

OUTS DX, [*segmentregister*:] *source*

OUTSB

OUTSW

OUTSD (80386 only)

INS [ES:] *destination*, DX

INSB

INSW

INSD (80386 only)

The INS and OUTS instructions require that the number of the port be in DX. The port cannot be specified as an immediate value, as it can be with IN and OUT.

To move the data, load the count into CX. The string to be transferred by INS is considered the destination string, so it must be pointed to by ES:DI. The string to be transferred by OUTS is considered the source string, so it must be pointed to by DS:SI.

If you specify the source or destination as an operand, DX must be specified. Otherwise DX is assumed and should be omitted.

If you need to process the string as it is transferred (for instance, to check for the end of a null terminated string), you must set up the loop yourself instead of using the REP instruction prefix.

```

count    .DATA
buffer   EQU      100
inport   DB      count DUP (?)
         DW      ?
         .CODE
         .
         .                ; Assume ES = DS
         .
         cld              ; Work upward
         mov     cx, count ; Load length to transfer
         mov     di, OFFSET buffer ; Load address of
                                   ; destination
         mov     dx, inport ; Load port number
         rep     insb      ; Transfer the string
                                   ; from port to buffer

```


Section 21

Calculating with a Math Coprocessor

The 8087 family coprocessors are used to do fast mathematical calculations. When used with real numbers, packed BCD numbers, or long integers, they do calculations many times faster than the same operations done with 8086 family processors. This subsection discusses the following topics:

- Coprocessor Architecture
- Emulation
- Using Coprocessor Instructions
- Coordinating Memory Access
- Transferring Data
- Doing Arithmetic Calculations
- Controlling Program Flow
- Using Transcendental Instructions
- Controlling the Coprocessor

This section explains how to use the 8087 family processors to transfer and process data. The approach taken is from an applications standpoint. Features that would be used by systems programmers (such the flags used when writing exception handlers) are not explained. This section is intended as a reference, not a tutorial.

Note: *This manual does not attempt to explain the mathematical concepts involved in using certain coprocessor features. It assumes that you will not need to use a feature unless you understand the mathematics involved. For example, you need to understand logarithms to use the FYL2X and FYL2XP1 instructions.*

Coprocessor Architecture

The math coprocessor works simultaneously with the main processor. However, since the coprocessor cannot handle device input or output, most data originates in the main processor.

The main processor and the coprocessor each have their own registers, which are completely separate and inaccessible to the other:

- Coprocessor Data Registers
- Coprocessor Control Registers

They exchange data through memory, since memory is available to both.

Ordinarily you follow these three steps when using the coprocessor:

- Load data from memory to coprocessor registers
- Process the data
- Store the data from coprocessor registers back to memory

Step 2, processing the data, can occur while the main processor is handling other tasks. Steps 1 and 3 must be coordinated with the main processor so that the processor and coprocessor do not try to access the same memory at the same time, as is explained in *Transferring Data*.

Coprocessor Data Registers

The 8087 family coprocessors have eight 80 bit data registers. Unlike 8086 family registers, the coprocessor data registers are organized as a stack. As data is pushed into the top register, previous data items move into higher numbered registers. Register 0 is the top of the stack; register 7 is the bottom. The syntax for specifying registers is shown below:

`ST[(number)]`

The number must be a digit between 0 and 7. If number is omitted, register 0 (top of stack) is assumed.

All coprocessor data are stored in registers in the temporary real format. This is the 10 byte IEEE format described in Section 8. The registers and the register format are shown in Figure 21-1.

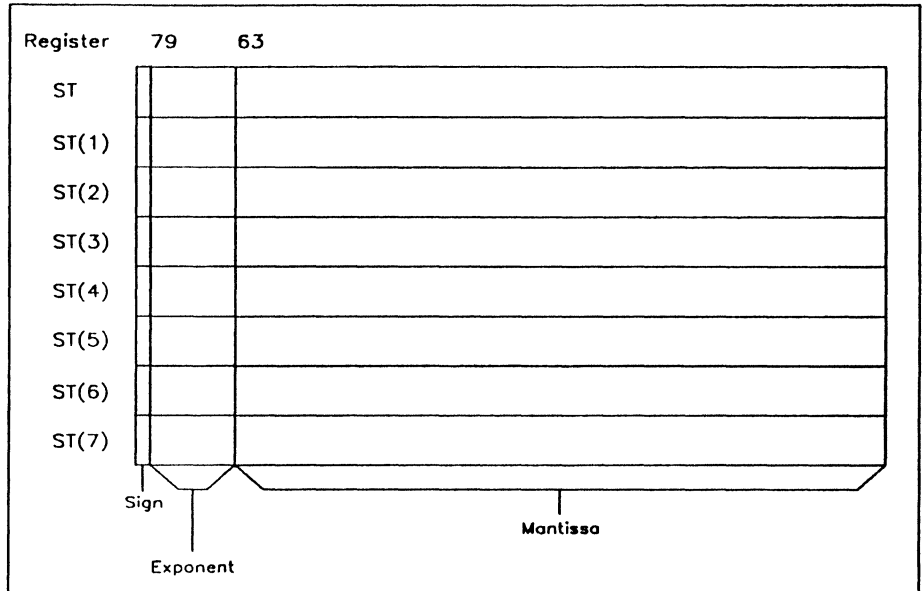


Figure 21–1. Coprocessor Data Registers

Internally, all calculations are done on numbers of the same type. Since temporary real numbers have the greatest precision, lower precision numbers are guaranteed not to lose precision as a result of calculations. The instructions that transfer values between the main processor and the coprocessor automatically convert numbers to and from the temporary real format.

Coprocessor Control Registers

The 8087 family coprocessors have seven 16 bit control registers. The most useful control registers are made up of bit fields or flags. Some flags control coprocessor operations, while others maintain the current status of the coprocessor. In this sense, they are much like the 8086 family flags registers.

You do not need to understand these registers to do most coprocessor operations. Control flags are set by default to the values appropriate for most programs. Errors and exceptions are reported in the status word

register. However, the coprocessor already has a default system for handling exceptions. Applications programmers can usually accept the defaults. Systems programmers may want to use the status word and control word registers when writing exception handlers, but such problems are beyond the scope of this manual.

Figure 21-2 shows the overall layout of the control registers including the control word, status word, tag word, instruction pointer, and operand pointer. The format of each of the registers is not shown, since these registers are generally of use only to systems programmers. The exception is the condition code bits of the status word register. These bits are explained in Controlling Program Flow.

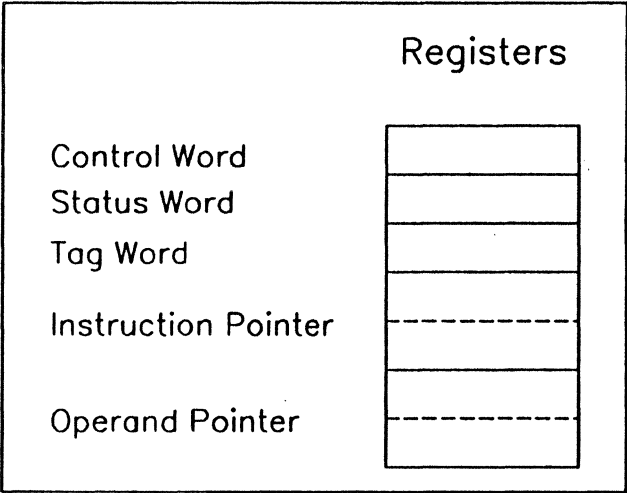


Figure 21-2. Coprocessor Control Register

Emulation

If you have a Microsoft high level language that supports floating point emulation, you can write assembly language procedures that use the emulator library when called from the high level language. First write the procedure by using coprocessor instructions, then assemble it using the `/E` option, and finally link it with your high level language modules. When compiling modules, use the compiler options that specify emulation.

Some coprocessor instructions are not emulated by Microsoft emulation libraries. Which instructions are emulated varies depending on the language and version. If you use a coprocessor instruction that is not emulated, the program will generate a run time error when it tries to execute the unemulated instruction. You cannot use a Microsoft emulation library with stand alone assembler programs, since the library depends on the compiler start up code.

See Section 4 for information on the `/E` option.

Using Coprocessor Instructions

This subsection explains how to use coprocessor instruction in the following ways:

- Using Implied Operands in the Classical Stack Form
- Using Memory Operands
- Specifying Operands in the Register Form
- Specifying Operands in the Register Pop Form

Coprocessor instructions are readily recognizable because, unlike all 8086 family instruction mnemonics, they start with the letter `F`.

Most coprocessor instructions have two operands, but in many cases one or both operands are implied. Often, one operand can be a memory operand; in this case, the other operand is always implied as the stack top register. Coprocessor instructions can never have immediate operands, and with the exception of the `FSTSW` instruction (see Loading Constants), they cannot have processor registers as operands. As with 8086 family instructions, memory to memory operations are never allowed. One operand must be a coprocessor register.

Instructions usually have a source and a destination operand. The source specifies one of the values to be processed. It is never changed by the operation. The destination specifies the value to be operated on and replaced with the result of the operation. If two operands are specified, the first is the destination and the second is the source.

The stack organization of registers gives the programmer flexibility to think of registers either as elements on a stack or as registers much like 8086 family registers.

Table 21–1 lists the variations of coprocessor instructions along with the syntax for each.

Table 21–1. Coprocessor Operand Forms

Instruction Form	Syntax	Implied Operands	Example
Classical stack	<i>Faction</i>	ST(1),ST	fadd
Memory	<i>Faction memory</i>	ST	fadd memloc
Register	<i>Faction</i> ST(<i>num</i>),ST <i>Faction</i> ST,ST(<i>num</i>)		fadd st(5),st fadd st,st(3)
Register pop	<i>FactionP</i> ST(<i>num</i>),ST		faddp st(4),st

Not all instructions accept all operand variations. For example, load and store instructions always require the memory form. Load constant instructions always take the classical stack form. Arithmetic instructions can usually take any form.

Some instructions that accept the memory form can have the letter I (integer) or B (BCD) following the initial F to specify how a memory operand is to be interpreted. For example, FILD interprets its operand as an integer and FBLD interprets its operand as a BCD number. If no type letter is included in the instruction name, the instruction works on real numbers.

Using Implied Operands in the Classical Stack Form

The classical stack form treats coprocessor registers like items on a stack. Items are pushed onto or popped off the top elements of the stack. Since only the top item can be accessed on a traditional stack, there is no need to specify operands. The first register (and the second if there are two operands) is always assumed.

In arithmetic operations (see *Doing Arithmetic Calculations*), the top of the stack (ST) is the source operand, and the second register (ST(1)) is the destination. The result of the operation goes into the destination operand, and the source is popped off the stack. The effect is that both of the values used in the operation are destroyed and the result is left at the top of the stack.

Instructions that load constants always use the stack form (see *Transferring Data to and from Registers*). In this case the constant created by the instruction is the implied source, and the top of the stack (ST) is the destination. The source is pushed into the destination.

Note: *The classical stack form with its implied operands is similar to the register pop form, not to the register form. For example, fadd, with the implied operands ST(1),ST, is equivalent to faddp st(1),st, rather than to fadd st(1),st*

fld1	; Push 1 into first position
fldpi	; Push pi into first position
fadd	; Add pi and 1 and pop

The status of the register stack after each instruction is shown in Figure 21–3.

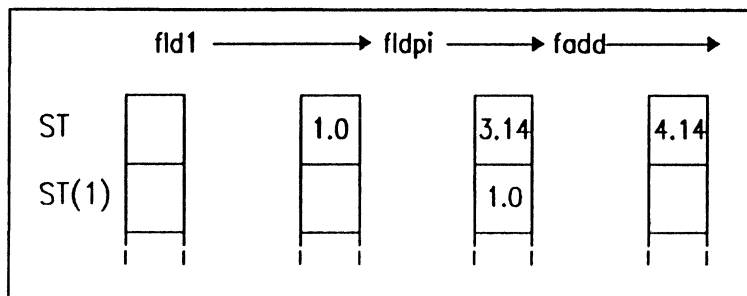


Figure 21–3. Status of Register Stack

Using Memory Operands

The memory form treats coprocessor registers like items on a stack. Items are pushed from memory onto the top element of the stack, or popped from the top element to memory. Since only the top item can be accessed on a traditional stack, there is no need to specify the stack operand. The top register (ST) is always assumed. However, the memory operand must be specified.

Memory operands can be used in load and store instructions (see Transferring Data to and from Registers). Load instructions push source values from memory to an implied destination register (ST). Store instructions pop source values from an implied source register (ST) to the destination in memory. Some versions of store instructions pop the register stack so that the source is destroyed. Others simply copy the source without changing the stack.

Memory operands can also be used in calculation instructions that operate on two values (see Doing Arithmetic Calculations). The memory operand is always the source. The stack top (ST) is always the implied destination. The result of the operation replaces the destination without changing its stack position.

```
m1      .DATA      1.0
m2      DD          2.0
        .CODE
        .
        .
        fld      m1      ; Push m1 into first position
        fld      m2      ; Push m2 into first position
        fadd     m1      ; Add m2 to first position
        fstp     m1      ; Pop first position into m1
        fst      m2      ; Copy first position to m2
```

The status of the register stack and the memory locations used in the instructions is shown in Figure 21–4.

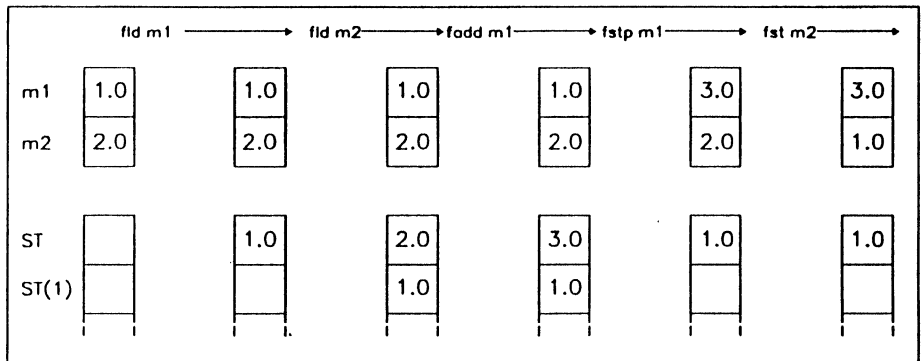


Figure 21-4. Status of Register Stack and Memory Locations

Specifying Operands in the Register Form

The register form treats coprocessor registers as traditional registers. Registers are specified the same as 8086 family instructions with two register operands. The only limitation is that one of the two registers must be the stack top (ST). In the register form, operands are specified by name. The second operand is the source; it is not affected by the operation. The first operand is the destination; its value is replaced with the result of the operation. The stack position of the operands does not change.

The register form can only be used with the FXCH instruction and with arithmetic instructions that do calculations on two values. With the FXCH instruction, the stack top is implied and need not be specified.

```
fadd    st(1),st    ; Add second position to first
                    ; result goes in second position
fadd    st,st(2)    ; Add first position to second
                    ; result goes in first position
fxch    st(1)       ; Exchange first and second
                    ; positions
```

The status of the register stack if the registers were previously initialized to 1.0, 2.0, and 3.0 is shown in Figure 21-5

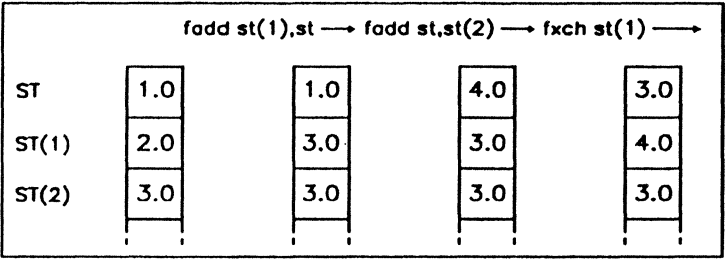


Figure 21–5. Register Stack with fxch Instruction

Specifying Operands in the Register Pop Form

The register pop form treats coprocessor registers as a modified stack. This form has some of the aspects of both a stack and registers. The destination register can be specified by name, but the source register must always be the stack top.

The result of the operation will be placed in the destination operand, and the stack top will be popped off the stack. The effect is that both values being operated on will be destroyed and the result of the operation will be saved in the specified destination register. The register pop form is only used for instructions that do calculations on two values.

```
faddp    st(2),st    ; Add first and third positions
                    ; and pop
                    ; first position destroyed
                    ; third moves to second and
                    ; holds result
```

The status of the register stack if the registers were already initialized to 1.0, 2.0, and 3.0 is shown in Figure 21–6.

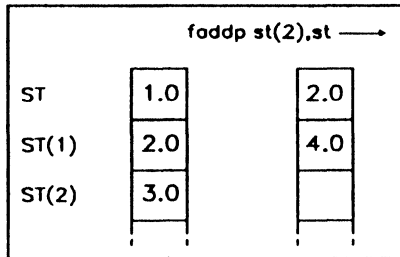


Figure 21–6. Register Stack with faddp Instruction

Coordinating Memory Access

Problems of coordinating memory access can occur when the coprocessor and the main processor both try to access a memory location at the same time. Since the processor and coprocessor work independently, they may not finish working on memory in the order in which you give instructions. There are two separate cases, and they are handled in different ways.

In the first case, if a processor instruction is given and then followed by a coprocessor instruction, the coprocessor must wait until the processor is finished before it can start the next instruction. This is handled automatically by MASM for the 8088 and 8086 or by the processor for the 80186, 80286, and 80386.

Note: To synchronize operations between the 8088 or 8086 processor and the 8087 coprocessor, each 8087 instruction must be preceded by a WAIT instruction. This is not necessary for the 80287 or 80387. If you use the .8087 directive, MASM inserts WAIT instructions automatically. However, if you use the .286 or .386 directive, MASM assumes the instructions are for the 80287 or 80387 and does not insert the WAIT instructions. If your code will never need to run on an 8086 or 8088 processor, you can make your programs shorter and more efficient by using the .286 or .386 directive.

In the second case, if a coprocessor instruction that accesses memory is followed by a processor instruction attempting to access the same memory location, memory access is not automatically synchronized. For instance, if you store a coprocessor register to a variable and then try to load that variable into a processor register, the coprocessor may not be finished. Thus the processor gets the value that was in memory before the coprocessor finished rather than the value stored by the coprocessor. Use the WAIT or FWAIT instruction (they are mnemonics for the same instruction) to ensure that the coprocessor finishes before the processor begins.

```
; Coprocessor instruction first – Wait needed
    fist      mem32          ; Store to memory
    fwait                    ; Wait until coprocessor
                           ; is done
    mov       ax,WORD PTR mem32 ; Move to register
    mov       dx,WORD PTR mem32[2]
; Processor instruction first – No wait needed
    mov       WORD PTR mem32,ax ; Load memory
    mov       WORD PTR mem32[2],dx
    fild      mem32            ; Load to register
```

Transferring Data

The 8087 family coprocessors have separate instructions for each of the following types of transfers:

- Transferring data between memory and registers
- Loading certain common constants into registers
- Transferring control data to and from memory

Transferring Data to and from Registers

Data transfer instructions transfer data between main memory and the coprocessor registers, or between different coprocessor registers. Two basic principles govern data transfers:

- The instruction determines whether a value in memory will be considered an integer, a BCD number, or a real number. The value is always considered a temporary real number once it is transferred to the coprocessor.
- The size of the operand determines the size of a value in memory. Values in the coprocessor always take up 10 bytes.

The adjustments between formats are made automatically. Notice that floating point numbers must be stored in the IEEE format, not in the Microsoft Binary format. Data are automatically stored correctly by default. It is stored incorrectly and the coprocessor instructions disabled if you use the `.MSFLOAT` directive. Data formats for real numbers are explained in Section 8.

Data are transferred to stack registers by using load commands. These push data onto the stack from memory or coprocessor registers. Data are removed by using store commands. Some store commands pop data off the register stack into memory or coprocessor registers, whereas others simply copy the data without changing it on the stack.

Real Transfers

The following instructions are available for transferring real numbers.

Syntax	Description
FLD <i>mem</i>	Pushes a copy of <i>mem</i> into ST. The source must be a 4, 8, or 10 byte memory operand. It is automatically converted to the temporary real format.
FLD ST(<i>num</i>)	Pushes a copy of the specified register into ST.
FST <i>mem</i>	Copies ST to <i>mem</i> without affecting the register stack. The destination can be a 4 or 8 byte memory operand. It is automatically converted from temporary real format to short real or long real format, depending on the size of the operand. It cannot be stored in the 10 byte real format.
FST ST(<i>num</i>)	Copies ST to the specified register. The current value of the specified register is replaced.
FSTP <i>mem</i>	Pops a copy of ST into <i>mem</i> . The destination can be a 4, 8, or 10 byte memory operand. It is automatically converted from temporary real format to the appropriate real number format, depending on the size of the operand.
FSTP ST(<i>num</i>)	Pops ST into the specified register. The current value of the specified register is replaced.
FXCH [ST(<i>num</i>)]	Exchanges the value in ST with the value in ST(<i>num</i>). If no operand is specified, ST(0) and ST(1) are exchanged.

Integer Transfers

The following instructions are available for transferring binary integers.

Syntax	Description
FILD <i>mem</i>	Pushes a copy of <i>mem</i> into ST. The source must be a 2, 4, or 8 byte integer memory operand. It is interpreted as an integer and converted to temporary real format.
FIST <i>mem</i>	Copies ST to <i>mem</i> . The destination must be a 2 or 4 byte memory operand. It is automatically converted from temporary real format to a word or a doubleword, depending on the size of the operand. It cannot be converted to a quadword integer.
FISTP <i>mem</i>	Pops ST into <i>mem</i> . The destination must be a 2, 4, or 8 byte memory operand. It is automatically converted from temporary real format to a word, doubleword, or quadword integer, depending on the size of the operand.

Packed BCD Transfers

The following instructions are available for transferring BCD integers.

Syntax	Description
FBLD <i>mem</i>	Pushes a copy of <i>mem</i> into ST. The source must be a 10 byte memory operand. It should contain a packed BCD value, although no check is made to see that the data is valid.
FBSTP <i>mem</i>	Pops ST into <i>mem</i> . The destination must be a 10 byte memory operand. The value is rounded to an integer if necessary, and converted to a packed BCD value.
fld <i>m1</i>	; Push <i>m1</i> into first item
fld <i>st(2)</i>	; Push third item into ; first
fst <i>m2</i>	; Copy first item to <i>m2</i>
fxch <i>st(2)</i>	; Exchange first and third ; items
fstp <i>m1</i>	; Pop first item into <i>m1</i>

With the assumption that registers ST and ST(1) were previously initialized to 3.0 and 4.0, the status of the register stack is shown in Figure 21-7.

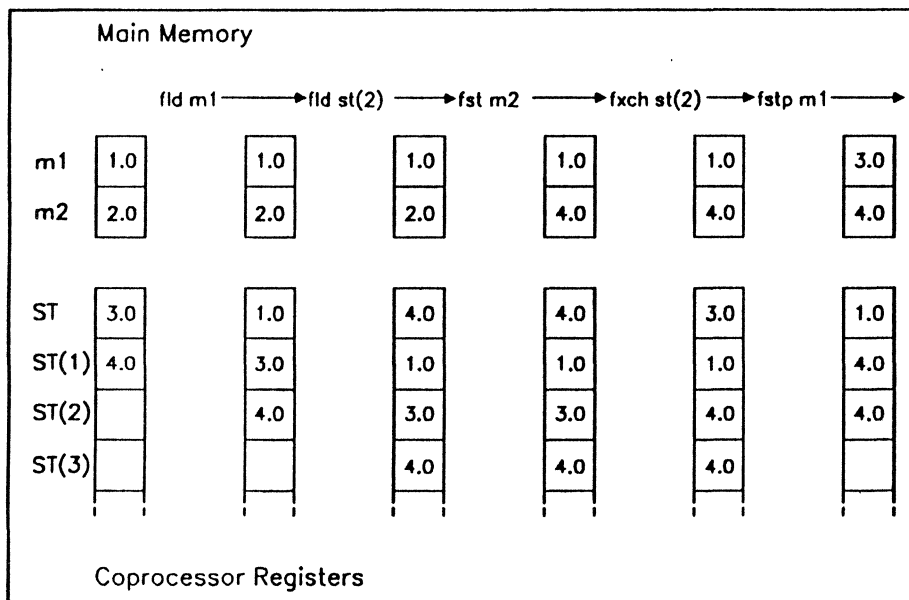


Figure 21-7. Register Stack Using Packed BCD

The following example illustrates one way of doing run time type conversions.

```

shortreal    .DATA
longreal     DD      100 DUP (?)
              DQ      100 DUP (?)
              .CODE
              .
              ; Assume array shortreal has
              ;   been
              ; filled by previous code
              .
              mov     cx,100      ; Initialize loop
              xor     si,si       ; Clear pointer into shortreal
              xor     di,di       ; Clear pointer into longreal
again:        fld     shortreal[si] ; Push shortreal
              fstp    longreal[di] ; Pop longreal
              add     si,4        ; Increment source pointer
              add     di,8        ; Increment destination
              ;   pointer
              loop    again      ; Do it again

```

Loading Constants

Constants cannot be given as operands and loaded directly into coprocessor registers. You must allocate memory and initialize the variable to a constant value. The variable can then be loaded by using one of the load instructions described in Transferring Data to and from Registers.

However, special instructions are provided for loading certain constants. You can load 0, 1, pi, and several common logarithmic values directly. Using these instructions is faster and often more precise than loading the values from initialized variables.

The instructions that load constants all have the stack top as the implied destination operand. The constant to be loaded is the implied source operand. The instructions are listed below.

Syntax	Description
FLDZ	Pushes 0 into ST
FLD1	Pushes 1 into ST
FLDPI	Pushes the value of pi into ST
FLDL2E	Pushes the value of $\log_2 e$ into ST
FLDL2T	Pushes $\log_2 10$ into ST
FLDLG2	Pushes $\log_{10} 2$ into ST
FLDLN2	Pushes $\log_e 2$ ST

Transferring Control Data

The coprocessor data area, or parts of it, can be stored to memory and later loaded back. One reason for doing this is to save a snapshot of the coprocessor state before going into a procedure, and restore the same status after the procedure. Another reason is to modify coprocessor behavior by storing certain data to main memory, operating on the data with 8086 family instructions, and then loading it back to the coprocessor data area.

You can choose to transfer the entire coprocessor data area, the control registers, or just the status or control word. Applications programmers seldom need to load anything other than the status word.

All the control transfer instructions take a single memory operand. Load instructions use the memory operand as the destination; store instructions use it as the source. The coprocessor data area is the implied source for load instructions and the implied destination for store instructions.

Each store instruction has two forms. The “wait form” checks for unmasked numeric error exceptions and waits until they have been handled. The “no–wait” form (which always begins with FN) ignores unmasked exceptions. The instructions are listed below.

Syntax	Description
FLDCW <i>mem2byte</i>	Loads control word
F \llcorner N \llcorner STCW <i>mem2byte</i>	Stores control word
F \llcorner N \llcorner STSW <i>mem2byte</i>	Stores status word
FLENV <i>mem14byte</i>	Loads environment
F \llcorner N \llcorner STENV <i>mem14byte</i>	Stores environment
FRSTOR <i>mem94byte</i>	Restores state
F \llcorner N \llcorner SAVE <i>mem94byte</i>	Saves state

Starting with the 80287, the FSTSW and FNSTSW instructions can store data directly to the AX register. This is the only case in which data can be transferred directly between processor and coprocessor registers, as shown below:

fstsw ax

In 32 bit mode, the 80387 stores 32 bit addresses in the instruction and operand pointers. Therefore, the FSAVE instruction stores 98 bytes instead of 94, and the FSTENV instruction stores 18 bytes instead of 14.

Doing Arithmetic Calculations

The math coprocessors offer a rich set of instructions for doing the following arithmetic:

- Addition
- Normal Subtraction
- Reversed Subtraction
- Multiplication
- Normal Division
- Reversed Division
- Other Operations

Most arithmetic instructions accept operands in any of the formats discussed in Using Coprocessor Instructions.

When using memory operands with an arithmetic instruction, make sure you indicate in the name whether you want the memory operand to be treated as a real number or an integer. For example, use FADD to add a real number to the stack top or FIADD to add an integer to the stack top. You do not need to specify the operand type in the instruction if both operands are stack registers, since register values are always real numbers.

You cannot do arithmetic on BCD numbers in memory. You must use FBLD to load the numbers into stack registers.

The arithmetic instructions are listed below.

Addition

The following instructions add the source and destination and put the result in the destination.

Syntax	Description
FADD	Classical stack form. Adds ST and ST(1) and pops the result into ST. Both operands are destroyed.
FADD ST(<i>num</i>),ST	Register form with stack top as source. Adds the two register values and replaces ST(<i>num</i>) with the result.
FADD ST,ST(<i>num</i>)	Register form with stack top as destination. Adds the two register values and replaces ST with the result.
FADD <i>mem</i>	Real memory form. Adds a real number in <i>mem</i> to ST. The result replaces ST.
FIADD <i>mem</i>	Integer memory form. Adds an integer in <i>mem</i> to ST. The result replaces ST.
FADDP ST(<i>num</i>),ST	Register pop form. Adds the two register values and pops the result into ST(<i>num</i>). Both operands are destroyed.

Normal Subtraction

The following instructions subtract the source from the destination and put the difference in the destination. Thus the number being subtracted from is replaced by the result.

Syntax	Description
FSUB	Classical stack form. Subtracts ST from ST(1) and pops the result into ST. Both operands are destroyed.
FSUB ST(<i>num</i>),ST	Register form with stack top as source. Subtracts ST from ST(<i>num</i>) and replaces ST(<i>num</i>) with the result.

FSUB ST,ST(<i>num</i>)	Register form with stack top as destination. Subtracts ST(<i>num</i>) from ST and replaces ST with the result.
FSUB <i>mem</i>	Real memory form. Subtracts the real number in <i>mem</i> from ST. The result replaces ST.
FISUB <i>mem</i>	Integer memory form. Subtracts the integer in <i>mem</i> from ST. The result replaces ST.
FSUBP ST(<i>num</i>),ST	Register pop form. Subtracts ST from ST(<i>num</i>) and pops the result into ST(<i>num</i>). Both operands are destroyed.

Reversed Subtraction

The following instructions subtract the destination from the source and put the difference in the destination. Thus the number subtracted is replaced by the result.

Syntax	Description
FSUBR	Classical stack form. Subtracts ST(1) from ST and pops the result into ST. Both operands are destroyed.
FSUBR ST(<i>num</i>),ST	Register form with stack top as source. Subtracts ST(<i>num</i>) from ST and replaces ST(<i>num</i>) with the result.
FSUBR ST,ST(<i>num</i>)	Register form with stack top as destination. Subtracts ST from ST(<i>num</i>) and replaces ST with the result.
FSUBR <i>mem</i>	Real memory form. Subtracts ST from the real number in <i>mem</i> . The result replaces ST.
FISUBR <i>mem</i>	Integer memory form. Subtracts ST from the integer in <i>mem</i> . The result replaces ST.
FSUBRP ST(<i>num</i>),ST	Register pop form. Subtracts ST(<i>num</i>) from ST and pops the result into ST(<i>num</i>). Both operands are destroyed.

Multiplication

The following instructions multiply the source and destination and put the product in the destination.

Syntax	Description
FMUL	Classical stack form. Multiplies ST by ST(1) and pops the result into ST. Both operands are destroyed.
FMUL ST(<i>num</i>),ST	Register form with stack top as source. Multiplies the two register values and replaces ST(<i>num</i>) with the result.
FMUL ST,ST(<i>num</i>)	Register form with stack top as destination. Multiplies the two register values and replaces ST with the result.
FMUL <i>mem</i>	Real memory form. Multiplies a real number in <i>mem</i> by ST. The result replaces ST.
FIMUL <i>mem</i>	Integer memory form. Multiplies an integer in <i>mem</i> by ST. The result replaces ST.
FMULP ST(<i>num</i>),ST	Register pop form. Multiplies the two register values and pops the result into ST(<i>num</i>). Both operands are destroyed.

Normal Division

The following instructions divide the destination by the source and put the quotient in the destination. Thus the dividend is replaced by the quotient.

Syntax	Description
FDIV	Classical stack form. Divides ST(1) by ST and pops the result into ST. Both operands are destroyed.
FDIV ST(<i>num</i>),ST	Register form with stack top as source. Divides ST(<i>num</i>) by ST and replaces ST(<i>num</i>) with the result.

FDIV ST,ST(<i>num</i>)	Register form with stack top as destination. Divides ST by ST(<i>num</i>) and replaces ST with the result.
FDIV <i>mem</i>	Real memory form. Divides ST by the real number in <i>mem</i> . The result replaces ST.
FIDIV <i>mem</i>	Integer memory form. Divides ST by the integer in <i>mem</i> . The result replaces ST.
FDIVP ST(<i>num</i>),ST	Register pop form. Divides ST(<i>num</i>) by ST and pops the result into ST(<i>num</i>). Both operands are destroyed.

Reversed Division

The following instructions divide the source by the destination and put the quotient in the destination. Thus the divisor is replaced by the quotient.

Syntax	Description
FDIVR	Classical stack form. Divides ST by ST(1) and pops the result into ST. Both operands are destroyed.
FDIVR ST(<i>num</i>),ST	Register form with stack top as source. Divides ST by ST(<i>num</i>) and replaces ST(<i>num</i>) with the result.
FDIVR ST,ST(<i>num</i>)	Register form with stack top as destination. Divides ST(<i>num</i>) by ST and replaces ST with the result.
FDIVR <i>mem</i>	Real memory form. Divides the real number in <i>mem</i> by ST. The result replaces ST.
FIDIVR <i>mem</i>	Integer memory form. Divides the integer in <i>mem</i> by ST. The result replaces ST.
FDIVRP ST(<i>num</i>),ST	Register pop form. Divides ST by ST(<i>num</i>) and pops the result into ST(<i>num</i>). Both operands are destroyed.

Other Operations

The following instructions all use the stack top (ST) as an implied destination operand. The result of the operation replaces the value in the stack top. No operand should be given.

Syntax	Description
FABS	Sets the sign of ST to positive
FCHS	Reverses the sign of ST.
FRNDINT	Rounds ST to an integer.
FSQRT	Replaces the contents of ST with its square root.
FSCALE	Scales by powers of two by adding the value of ST(1) to the exponent of the value in ST. This effectively multiplies the stack top value by two to the power contained in ST(1). Since the exponent field is an integer, the value in ST(1) should normally be an integer.
FPREM	Calculates the partial remainder by performing modulo division on the top two stack registers. The value in ST is divided by the value in ST(1). The remainder replaces the value in ST. The value in ST(1) is unchanged. Since this instruction works by repeated subtractions, it can take a lot of execution time if the operands are greatly different in magnitude. FPREM is sometimes used with trigonometric functions.
EXTRACT	Breaks a number down into its exponent and mantissa and pushes the mantissa onto the register stack. Following the operation, ST contains the value of the original mantissa and ST(1) contains the value of the unbiased exponent.

Note: *The 80387 has a new instruction called **FPREM1**. Its effect is similar to that of **FPREM**, but it conforms to the IEEE standard.*

The following example solves quadratic equations. It does no error checking and fails for some values because it attempts to find the square root of a negative number. You could enhance the code by using the FTST instruction (see Comparing Operands to Control Program) to check for a negative number or 0 just before the square root is calculated. If b^2 minus $4ac$ is negative or 0, the code can jump to routines that handle special cases for no solution or one solution, respectively.

```

        .DAT
a        DD        3.0
b        DD        7.0
c        DD        2.0
posx     DD        0.0
negx     DD        0.0

        .CODE
        .
        .
        .
; Solve quadratic equation – no error checking

        fldl                ; Get constants 2 and 4
        fadd    st,st       ; 2 at bottom
        fld    st           ; Copy it
        fmul    a           ; = 2a

        fmul    st(1),st    ; = 4a
        fxch
        fmul    c           ; = 4a

        fld    b            ; Load b
        fmul    st,st       ; = b^2
        fsubr                ; = b^2 - 4ac
                                ; Negative value here produces
                                ; error
        fsqrt                ; = square root(b^2 - 4ac)
        fld    b            ; Load b
        fchs                ; Make it negative
        fxch
        fld    st           ; Copy square root
        fadd    st,st(2)    ; Plus version = -b + root((b^2 -
                                ; 4ac)
        fxch
        fsubp    st(2),st   ; Minus version = -b - root((b^2 -
                                ; 4ac)

        fdiv    st,st(2)    ; Divide plus version
        fstp    posx        ; Store it
        fdivr   negx        ; Divide minus version
        fstp    negx        ; Store it

```

Controlling Program Flow

The math coprocessors provide two main ways of controlling program flow:

- Comparing Operands
- Testing Control Flags after Other Instructions

There are several instructions that set control flags in the status word. The 8087 family control flags can be used with conditional jumps to direct program flow in the same way that 8086 family flags are used.

Since the coprocessor does not have jump instructions, you must transfer the status word to memory so that the flags can be used by 8086 family instructions.

An easy way to use the status word with conditional jumps is to move its upper byte into the lower byte of the processor flags. For example, use the following statements:

```
fstsw    mem16        ; Store status word in memory
fwait                    ; Make sure coprocessor is done
mov      ax,mem16      ; Move to AX
sahf                    ; Store upper word in flags
```

As noted in Transferring Control Data, you can save several steps by loading the status word directly to AX on the 80287 and 80387.

Figure 21–8 shows how the coprocessor control flags line up with the processor flags. C3 overwrites the zero flag, C2 overwrites the parity flag, and C0 overwrites the carry flag. C1 overwrites an undefined bit, so it cannot be used directly with conditional jumps, although you can use the TEST instruction to check C1 in memory or in a register. The sign and auxiliary carry flags are also overwritten, so you cannot count on them being unchanged after the operation.

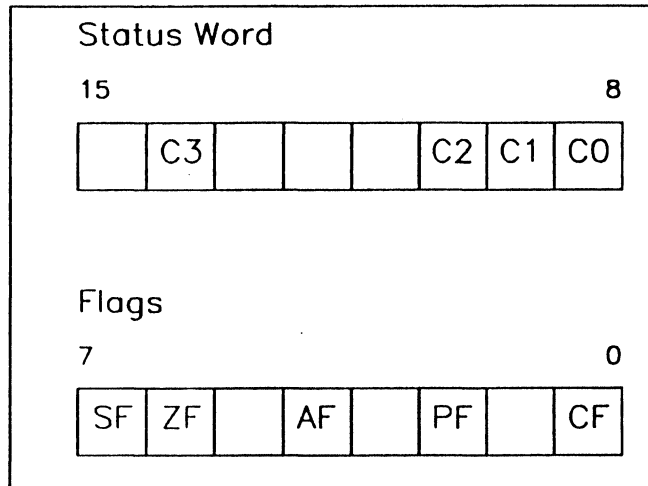


Figure 21–8. Coprocessor and Processor Control Flags

See Section 19 for more information on using conditional jump instructions based on flag status.

Comparing Operands to Control Program Flow

The 8087 family coprocessors provide several instructions for comparing operands. All these instructions compare the stack top (ST) to a source operand, which may either be specified or implied as ST(1). The compare instructions affect the C3, C2, and C0 control flags. The C1 flag is not affected.

Table 21–2 shows the flags set for each possible result of a comparison or test.

Table 21–2. Control Flag Settings after Compare or Test

After FCOM	After FTEST	C3	C2	C0
ST > source	ST is positive	0	0	0
ST < source	ST is negative	0	0	1
ST = source	ST is 0	1	0	0
Not comparable	ST is NAN or projective infinity	1	1	1

Variations on the compare instructions allow you to pop the stack once or twice, and to compare integers and zero. For each instruction, the stack top is always the implied destination operand. If you do not give an operand, ST(1) is the implied source. Some compare instructions allow you to specify the source as a memory or register operand. The compare instructions are listed below.

Compare

These instructions compare the stack top to the source. The source and destination are unaffected by the comparison.

Syntax	Description
FCOM	Compares ST to ST(1).
FCOM ST(<i>num</i>)	Compares ST to ST(<i>num</i>).
FCOM <i>mem</i>	Compares ST to <i>mem</i> . The memory operand can be a four or eight byte real number.
FICOM <i>mem</i>	Compares ST to <i>mem</i> . The memory operand can be a two or four byte integer.
FTST	Compares the ST to 0. The control registers will be affected as if ST had been compared to 0 in ST(1). Table 21–2 above shows the possible results.

Compare and Pop

These instructions compare the stack top to the source, and then pop the stack. Thus the destination is destroyed by the comparison.

Syntax	Description
FCOMP	Compares ST to ST(1) and pops ST off the register stack.
FCOMP ST(<i>num</i>)	Compares ST to ST(<i>num</i>) and pops ST off the register stack.
FCOMP <i>mem</i>	Compares ST to <i>mem</i> and pops ST off the register stack. The operand can be a four or eightbyte real number.
FICOMP <i>mem</i>	Compares ST to <i>mem</i> and pops ST off the register stack. The operand can be a two or four byte integer.
FCOMPP	Compares ST to ST(1), and then pops the stack twice. Both the source and destination are destroyed by the comparison.

Note: *Unordered compare instructions are available with the 80387. The FUCOM, FUCOMP, and FUCOMPP instructions are like FCOM, FCOMP, and FCOMPP except that the unordered versions do not cause invalid operation exceptions if one of the operands is a quiet NAN (not a number). Exceptions and NANs are beyond the scope of this manual and are not explained here. See Intel coprocessor reference books for more information.*

In the following example, notice how conditional blocks are used to enhance 80287 code. If you define the symbol c287 from the command line by using the /Dsymbol option (see Section 4) the code is smaller and faster, but does not run on an 8087.

```

                                IFDEF      c287
                                .287
                                ENDIF
                                .DATA
down      DD      10.35      ; Sides of a rectangle
across    DD      13.07
diameter  DD      12.93      ; Diameter of a circle
status    DW      ?

```

```
.CODE
:
:
; Get area of rectangle
    fld     across      ; Load one side
    fmul    down        ; Multiply by the other

; Get area of circle
    fldl
    fadd     st,st      ; Load one and
                        ; double it to get constant 2
    fdivr    diameter   ; Divide diameter to get radius
    fmul     st,st      ; Square radius
    fldpi
    fmul
                        ; Load pi
                        ; Multiply it

; Compare area of circle and rectangle
    fcompp
    IFNDEF   c287        ; Compare and throw both away
    fstsw    status     ; Load from coprocessor to memory
    fwait
                        ; Wait for coprocessor
    mov      ax,status   ; Memory to register
    ELSE
    fstsw    ax          ; (for 287+, skip memory)
    ENDIF
    sahf
                        ; to flags
    jp       nocomp      ; If parity set, can't compare
    jz       same        ; If zero set, they're the same
    jc       rectangle   ; If carry set, rectangle is
                        ; bigger
    jmp      circle      ; else circle is bigger

nocomp:    .            ; Error handler
:
same:      .            ; Both equal
:
rectangle: .            ; Rectangle bigger
:
circle:    .            ; Circle bigger
```

Testing Control Flags after Other Instructions

In addition to the compare instructions, the FXAM and FPREM instructions affect coprocessor control flags.

The FXAM instruction sets the value of the control flags based on the type of the number in the stack top (ST). This instruction is used to identify and handle special values such as infinity, zero, unnormal numbers, denormal numbers, and NaNs (not a number). Certain math operations

are capable of producing these special format numbers. A description of them is beyond the scope of this manual. The possible settings of the flags are shown in the Microsoft Macro Assembler Reference.

FPREM also sets control flags. Since this instruction must sometimes be repeated to get a correct remainder for large operands, it uses the C2 flag to indicate whether the remainder returned is partial (C2 is set) or complete (C2 is clear). If the bit is set, the operation should be repeated.

FPREM also returns the least significant three bits of the quotient in C0, C3, and C1. These bits are useful for reducing operands of periodic transcendental functions, such as sine and cosine, to an acceptable range. The technique is not explained here. The possible settings for each flag are shown in the Microsoft Macro Assembler Reference.

Using Transcendental Instructions

The 8087 family coprocessors provide a variety of instructions for doing transcendental calculations, including exponentiation, logarithmic calculations, and some trigonometric functions.

Use of these advanced instructions is beyond the scope of this manual. However, the instructions are listed below for reference. All transcendental instructions have implied operands—either ST as a single destination operand, or ST as the destination and ST(1) as the source.

Instruction	Description
F2XM1	Calculates $2^x - 1$, where x is the value of the stacktop. The value x must be between 0 and .5, inclusive. Returning $2^x - 1$ instead of 2^x allows the instruction to return the value with greater accuracy. The programmer can adjust the result to get 2^x .
FYL2X	Calculates $Y \times \log_2 X$, where X is in ST and Y is in ST(1). The stack is popped, so both X and Y are destroyed, leaving the result in ST. The value of X must be positive.

FYL2XP1	Calculates Y times $\log_2(X + 1)$, where X is in ST and Y is in ST(1). The stack is popped, so both X and Y are destroyed, leaving the result in ST. The absolute value of X must be between 0 and the square root of 2 divided by 2. This instruction is more accurate than FYL2X when computing the log of a number close to 1.
FPTAN	Calculates the tangent of the value in ST. The result is a ratio Y/X , with Y replacing the value in ST and X pushed onto the stack so that after the instruction, ST contains Y and ST(1) contains X . The value being calculated must be a positive number less than $\pi/4$. The result of the FPTAN instruction can be used to calculate other trigonometric functions, including sine and cosine.
FPATAN	Calculates the arctangent of the ratio Y/X , where X is in ST and Y is in ST(1). The stack is popped, so both X and Y are destroyed, leaving the result in ST. Both X and Y must be positive numbers less than infinity, and Y must be less than X . The result of the FPATAN instruction can be used to calculate other inverse trigonometric functions, including arcsine and arccosine.

The following additional trigonometric functions are available on the 80387.

Instruction	Description
FSIN	Calculates the sine of the value in ST. The stack top value is replaced by its sine.
FCOS	Calculates the cosine of the value in ST. The stack top value is replaced by its cosine.
FSINCOS	Calculates the sine and cosine of the value in ST. When the instruction is complete, the value in ST is the cosine of the original stack top value. The value in ST(1) is the sine of the original stack top value. One of the values is pushed so that the former value in ST(1) is in ST(2).

Controlling the Coprocessor

Additional instructions are available for controlling various aspects of the coprocessor. With the exception of FINIT, these instructions are generally used only by systems programmers. They are summarized below, but not fully explained or illustrated. Some instructions have a wait version and a no wait version. The no wait versions have N as the second letter.

Syntax	Description
F[N]INIT	Resets the coprocessor and restores all the default conditions in the control and status words. It is a good idea to use this instruction at the start and end of your program. Placing it at the start ensures that no register values from previous programs affect your program. Placing it at the end ensures that register values from your program will not affect later programs.
F[N]CLEX	Clears all exception flags and the busy flag of the status word. It also clears the error status flag on the 80287 and 80387, or the interrupt request flag on the 8087.
FINCSTP	Adds one to the stack pointer in the status word. Do not use to pop the register stack. No tags or registers are altered.
FDECSTP	Subtracts one from the stack pointer in the status word. No tags or registers are altered.

FREE ST(<i>num</i>)	Marks the specified register as empty.
FNOP	Copies the stack top to itself, thus padding the executable file and taking up processing time without having any effect on registers or memory.

Note: *The 8087 has the instructions FDISI, FNDISI, FENI, and FNENI. These instructions can be used to enable or disable interrupts. The 80287 and 80387 coprocessors permit these instructions, but ignore them. Applications programmers will not normally need these instructions. Systems programmers should avoid using them so that their programs are portable to all coprocessors.*

Starting with the 80287, the FSETPM (Set Protected Mode) instruction is available. This instruction enables the coprocessor to run in protected mode. The primary difference is that the addresses stored in the instruction and operand pointers have a segment selector instead of an actual segment address. See Section 15 for information on segment selectors.

Either the .286P or .386P directive must be given before the FSETPM instruction can be used. Protected mode operating systems normally set protected mode automatically. Therefore, you need this instruction only if you are writing control software.

Section 22

Controlling the Processor

The 8086 family processors provide instructions for processor control. Some of these instructions are available on all processors; others are for controlling protected mode operations on the 80286 and 80386.

System control instructions have limited use in applications programming. They are primarily used by systems programmers who write operating systems and other control software. Since systems programming is beyond the scope of this manual, the systems control instructions are summarized, but not explained in detail, in the sections below.

Controlling Timing and Alignment

The NOP instruction does nothing but take up time and space. It works by exchanging the AX register with itself. The NOP instruction can be used for delays in timing loops, or to pad executable code for alignment.

Normally, applications programmers should avoid using the NOP instruction in timing loops, since such loops take different lengths of time on different machines. A better way to control timing is to use the DOS time function, since it is based on the computer's internal clock rather than on the speed of the processor.

MASM automatically inserts NOP instructions for padding when you use the ALIGN or EVEN directive to align data or code on a given boundary.

Controlling the Processor

The WAIT, ESC, LOCK, and HLT instructions control different aspects of the processor.

These instructions can be used to control processes handled by external coprocessors. The 8087 family coprocessors are the coprocessors most commonly used with 8086 family processors, but 8086 based machines can work with other coprocessors if they have the proper hardware and control software. These instructions are summarized below:

Instruction	Description
LOCK	Locks out other processors until a specified instruction is finished. This is a prefix that precedes the instruction. It can be used to make sure that a coprocessor does not change data being worked on by the processor.
WAIT	Instructs the processor to do nothing until it receives a signal that a coprocessor has finished with a task being performed at the same time. See Coordinating Memory Access for information on using WAIT or its coprocessor equivalent, FWAIT, with the 8087 family coprocessors.
ESC	Provides an instruction and possibly a memory operand for use by a coprocessor. MASM automatically inserts ESC instructions when required for use with 8087 family coprocessors.
HLT	Stops the processor until an interrupt is received. It can be used in place of an endless loop if a program needs to wait for an interrupt.

Controlling Protected Mode Processes

Protected mode is available starting with the 80286 processor. This mode is generally initiated and controlled by an operating system.

The instructions that control protected mode are privileged and can only be used if the .286P or .386P directives have been given. These instructions are generally needed only for operating systems and other control software. Although used for managing protected mode, some of these instructions can be used when the processor is not in protected mode.

Note that, under protected mode operating systems, applications programmers do not need to use protected mode instructions. Process control is managed through system calls.

Some privileged mode instructions use internal registers of the 80286 or 80386 processors. Instructions are provided for loading values from these registers into memory where the values can be modified. Other instructions can then be used to store the values back to the special registers.

The privileged mode instructions are listed below:

Instruction	Description
LAR	Loads access rights
LSL	Loads segment limit
LGDT	Loads global descriptor table
SGDT	Stores global descriptor table
LIDT	Loads 8 byte interrupt descriptor table
SIDT	Stores 8 byte interrupt descriptor table
LLDT	Loads local descriptor table
SLDT	Stores local descriptor table
LTR	Loads task register
STR	Stores task register

LMSW	Loads machine status word
SMCW	Stores machine status word
ARPL	Adjusts requested privilege level
CLTS	Clears task switched flag
VERR	Verifies read access
VERW	Verifies write access

Controlling the 80386

The 80386 processor can use all the privileged mode instructions of the 80286, but it also allows you to use MOV to transfer data between general purpose registers and special registers.

The following special registers can be accessed with move instructions on the 80386:

Type	Registers
Control	CR0, CR2, and CR3
Debug	DR0, DR1, DR2, DR3, DR6, and DR7
Test	TR6 and TR7

These registers can be moved directly to 32 bit registers or from them.

```
mov    eax,cr0      ; Load CR0 into EAX
mov    cr1,ecx       ; Store ECX in CR1
```

Appendix A

New Features

The CTOS Microsoft Macro Assembler package has many significant new features. Some of the most important are the support for the 80386 processor and an optional simplified system of defining segments. This appendix describes these features and tells you where they are documented.

MASM Enhancements

MASM, the Macro Assembler program, now has several important enhancements over Version 4.0 and other previous versions. The sections below summarize new options, directives, instructions, and other features.

80386 Support

MASM now supports the 80386 instruction set and addressing modes. The 80386 processor is a superset of other 8086 family processors. Most new features are simply 32 bit extensions of 16 bit features.

If you understand the features of the 16 bit 8086 family processors, then using the 32 bit extensions is not difficult. The new 32 bit registers are used in much the same way as the 16 bit registers. The 80386 registers are explained in Section 15.

However, some features of the 80386 processor are significantly different. These differences are noted throughout the manual. Areas of particular importance include the .386 directive for initializing the 80386. Section 6 discusses the USE32 and USE16 segment types for setting the segment word size. Section 7 discusses indirect addressing modes. Section 16 discusses 80386 indirect memory operands.

The 80386 processor and the 80387 coprocessor also have the new instructions listed in Table A-1.

Table A-1. 80386 and 80387 Instruction

Name	Mnemonic	Reference
Bit Scan Forward	BSF	Section 18
Bit Scan Reverse	BSR	Section 18
Bit Test	BT	Section 19
Bit Test and Complement	BTC	Section 19
Bit Test and Reset	BTR	Section 19
Bit Test and Set	BTS	Section 19
Move with Sign Extend	MOVSX	Section 17
Move with Zero Extend	MOVZX	Section 17
Set Byte on Condition	SETcondition	Section 19
Double Precision Shift Left	SHLD	Section 18
Double Precision Shift Right	SHRD	Section 18
Move to/from Special Registers	MOV	Section 20
Sine	FSIN	Section 21
Cosine	FCOS	Section 21
Sine Cosine	FSINCOS	Section 21
IEEE Partial Remainder	FPREM1	Section 21
Unordered Compare Real	FUCOM	Section 21
Unordered Compare Real and Pop	FUCOMP	Section 21
Unordered Compare Real and Pop Twice	FUCOMPP	Section 21

Segment Simplification

A new system of defining segments is available in MASM. The simplified segment directives use the Microsoft naming conventions. If you are willing to accept these conventions, segments can be defined more easily and consistently. However, this feature is optional. You can still use the old system if you need more direct control over segments or if you need to be consistent with existing code. See Section 7.

Performance Improvements

The performance of MASM has been enhanced in two ways: faster assembly and larger symbol space. The improvement varies depending on the relative amounts of code and data in the source file, and on the complexity of expressions used. Symbol space is now limited only by the amount of system memory available to your machine.

Enhanced Error Handling

Error handling has been enhanced in the following ways:

- Messages are divided into three levels: severe errors, serious warnings, and advisory warnings. The level of warning can be changed with the /W option. Type checking errors are now serious warnings rather than severe errors. See Section 4.
- During assembly, messages are output to the standard output device (by default, the screen). They can be redirected to a file or device. See Section 4.

New Options

The following command line options have been added:

Option	Description
<code>/W[0 1 2]</code>	Sets the warning level to determine what type of messages will be displayed. The three kinds are severe errors, serious warnings, and advisory warnings. See Section 4.
<code>/ZD</code>	The <code>/ZD</code> option causes the assembler to output line number information. See Section 4.
<code>/H</code>	Displays the MASM command line and options. See Section 4.
<code>/Dsym [=val]</code>	Allows definition of a symbol from the command line. This is an enhancement of a current option. See Section 4.

In addition, the new directives `.ALPHA` and `.SEQ` have been added; these directives have the same effect as the `/A` and `/S` options. See Section 7.

Environment Variables

MASM now supports two environment variables: `MASM` for specifying default options, and `INCLUDE` for specifying the search path for include files. See Section 4.

String Equates

String equates have been enhanced for easier use. By enclosing the argument to the `EQU` directive in angle brackets, you can ensure that the argument is evaluated as a string equate rather than as an expression. See Section 13 for more information on string equates. The expression operator `()` can now be used with macro arguments that are text macros as well as arguments that are expressions. See Section 13.

RETF and RETN Instructions

The RETF (Return Far) and RETN (Return Near) instructions are now available. These instructions enable you to define procedures without the PROC and ENDP directives. See Section 19.

Communal Variables

MASM now allows you to declare communal variables. These uninitialized global data items can be used in include files. They are compatible with variables declared in C include files. See Section 10.

Flexible Structure Definitions

Structure definitions can now include conditional assembly statements, thus enabling more flexible structures. See Section 9.

Appendix B

Error Messages and Exit Codes

This appendix lists and explains the messages and exit codes that can be generated by MASM and CREF. Messages are sent to the standard output device. By default, this device is the screen, but you can redirect the messages to a file or to a device such as a printer.

MASM Messages and Exit Codes

The assembler can display several kinds of messages as well as output an exit code; the kind of exit code output depends on the error, if any, encountered during the assembly.

Assembler Status Messages

After every assembly, MASM reports on the symbol space, errors, and warnings. A sample display is shown below:

```
CTOS Microsoft (R) Macro Assembler Version 5.1.1  
Copyright (C) Unisys Corp 1991. All rights reserved.
```

```
47904 + 353887 Bytes symbol space free
```

```
0 Warning Errors  
0 Severe Errors
```

The first line indicates how much near and far symbol space was unused during the assembly. This data may help you determine whether increasing the size of your program will exhaust available memory.

The first number indicates near symbol space. There is 64K total. The second number indicates far symbol space. This is equal to the amount of available short lived memory plus any memory remaining in the far heap. When far space is exhausted, additional symbols go into near space. Using both far and near space causes a decrease in speed of assembly.

You can use the `/V` option to direct MASM to display additional statistics. The number of source lines, the total number of source and include file lines, and the number of symbols are shown. This information appears only if no severe errors are encountered. An example is shown below:

```
742 Source Lines
799 Total Lines
44 Symbols
```

The `/T` option can be used to suppress all output to the screen after assembly.

Numbered Assembler Messages

The assembler displays messages on the screen whenever it encounters an error while processing a source file. It also displays a warning message whenever it encounters questionable syntax. Messages that can be associated with a particular line of code are numbered. General errors related to the entire assembly rather than to a particular line are unnumbered (see Unnumbered Error Messages).

Numbered error messages are displayed in the following format:

sourcefile(line) : code: message

The *sourcefile* is the name of the source file where the error occurred. If the error occurred in a macro in an include file, the *sourcefile* is the file where the macro was called and expanded—not the file where it was defined.

The *line* indicates the point in the source file where MASM was no longer able to assemble.

The code is an identifying code in the format used by all Microsoft language programs. It starts with the word error or warning followed by a five character code. The first character is a letter indicating the program or language. Assembler messages start with A. The first digit indicates the warning level. The number is 2 for severe errors, 4 for serious warnings, and 5 for advisory warnings. The next three digits are the error number. For example, severe error 38 is shown as A2038

The message is a descriptive line describing the error.

MASM messages are listed in numerical order in this section with a short explanation for each.

Code	Message
0	<p>Block nesting error</p> <p>Nested procedures, segments, structures, macros, or repeat blocks were not properly terminated. This error may indicate that you closed an outer level of nesting with inner levels still open.</p>
1	<p>Extra characters on line</p> <p>Sufficient information to define a statement has been received on a line, but additional characters were also provided. This may indicate that you provided too many arguments.</p>
2	<p>Internal error – Register already defined <i>symbol</i></p> <p>Note the conditions when the error occurs and report them to Unisys Corporation.</p>
3	<p>Unknown type specifier</p> <p>An invalid type specifier was used to give the size of a label or external declaration. For instance, BYTE or NEAR might have been misspelled.</p>
4	<p>Redefinition of symbol</p> <p>A symbol was defined in two places with different types. This error occurs during Pass 1 on the second declaration of the symbol.</p>
5	<p>Symbol is multidefined:</p> <p>A symbol is defined in two places. This error occurs during Pass 2 on each declaration of the symbol.</p>

- 6 Phase error between passes
- An ambiguous instruction or directive caused the relative address of a label to be changed between Pass 1 and Pass 2. You can use the /D option to produce a Pass 1 listing to aid in resolving phase errors between passes. The format of Pass 1 listings is discussed in Section 2.
- 7 Already had ELSE clause
- More than one ELSE clause was used within a conditional assembly block. Each nested ELSE must have its own IF directive and ENDIF.
- 8 Must be in conditional block
- An ENDIF or ELSE was specified without a corresponding IF directive.
- 9 Symbol not defined:
- A symbol was used without being defined. This error is produced for forward references on the first pass and is ignored if the references are resolved on the second pass.
- 10 Syntax error
- A statement did not match any recognizable assembler syntax. MASM tries to be specific, so this error only occurs if the statement bears no resemblance to any legal statement.
- 11 Type illegal in context
- The type specifier was given with an unacceptable size. For example, a procedure was defined as having BYTE type, instead of NEAR or FAR type.
- 12 Group name must be unique
- A name assigned as a group name was already defined as another type of symbol.

- 13 Must be declared during Pass 1: *symbol*
- An item was referenced before it was defined in Pass 1. For example, IF DEBUG is illegal if the symbol DEBUG was not previously defined.
- 14 Illegal public declaration
- A symbol was declared public illegally. For instance, a text equate cannot be declared public. Section 8 explains public declarations.
- 15 Symbol already different kind: *symbol*
- A symbol was redefined to a different kind of symbol. For example, a segment name was reused as a variable name, or a structure name was reused as an equate name.
- 16 Reserved word used as symbol: *name*
- An assembler keyword was used as a symbol. This is a warning, not an error, and can be ignored if you wish. However, the keyword is no longer available for its original purpose. For example, if you name a macro add, it replaces the ADD instruction.
- 17 Forward reference illegal
- A symbol was referenced before it was defined on Pass 1. For example, the following lines produce an error:
- ```

 DB count DUP(?)
count EQU 10

```
- The statements would be legal if the lines were reversed.
- 18            Operand must be register: *operand*
- A register was expected as an operand, but a symbol or constant was supplied.

20 Operand must be segment or group

A segment or group name was expected, but some other kind of operand was given. For instance, the ASSUME directive requires that the symbol assigned to a segment register be a segment name, a group name, a SEG expression, or a text equate representing a segment or group name. Thus the following statement is accepted:

```
ASSUME ds:SEG variable ; Legal
```

However, if the same statement is assigned to an equate, it is not accepted, as shown below:

```
segvar EQU SEG variable
ASSUME ds:segvar ; Illegal
```

22 Operand must be type specifier

An operand was expected to be a type specifier, such as NEAR or FAR, but some other kind of operand was received.

23 Symbol already defined locally

A symbol that had already been defined within the current module was declared EXTRN

24 Segment parameters are changed

A segment declaration with the same name as a previous segment declaration was given with arguments that did not match the previous declaration. See Section 5 for information on defining segments.

25 Improper align/combine type

SEGMENT parameters are incorrect. Check the align and combine types to make sure you have entered valid types from among those discussed in Section 5.

- |    |                                                                                                                    |
|----|--------------------------------------------------------------------------------------------------------------------|
| 26 | Reference to multidefined symbol                                                                                   |
|    | An instruction referenced a symbol defined in more than one place.                                                 |
| 27 | Operand expected                                                                                                   |
|    | An operand was expected, but an operator was received.                                                             |
| 28 | Operator expected                                                                                                  |
|    | An operator was expected, but an operand was received.                                                             |
| 29 | Division by 0 or overflow                                                                                          |
|    | An expression resulted in division by 0 or in a number too large to be represented.                                |
| 30 | Negative shift count                                                                                               |
|    | An expression using the SHR or SHL operator evaluated to a negative shift count.                                   |
| 31 | Operand types must match                                                                                           |
|    | An instruction received operands of different sizes. For example, this warning is generated by the following code: |

```

string DB This is a test
 .
 .
 .
 mov ax,string[4]

```

Since this is a warning rather than an error, MASM attempts to generate code based on its best guess of the intended result. If one of the operands is a register, the register size overrides the size of the other operand. In the example, the word size of AX overrides the byte size of string[4]. You can avoid this warning and make your code less ambiguous by specifying the operand size with the PTR operator. For example:

```

move ax,WORD PTR string[4]

```



32      Illegal use of external

An external variable was used incorrectly. See Section 8 for information about correct declaration and use of external symbols.

34      Operand must be record or field name

An operand was expected to be a record name or record field name, but another kind of operand was received.

35      Operand must have size

An operand was expected to have a specified size, but no size was supplied. For example, the following statement is illegal:

```
inc [bx]
```

Often this error can be remedied by using the PTR operator to specify a size type, as shown below:

```
inc BYTE PTR [bx]
```

38      Left operand must have segment

The left operand of a segment override expression must be a segment register, group, or segment name. For example, if mem1 and mem2 are variables, the following statement is illegal:

```
mov dx,mem1:mem2
```

39      One operand must be constant

The addition operator was used incorrectly. For instance, two memory operands cannot be added in an expression. Valid uses of the addition operator are explained in Section 9.

40      Operands must be in same segment, or one must be constant

The subtraction operator was used incorrectly. For instance, a memory operand in the code segment cannot be subtracted from a memory operand in the data segment. Valid uses of the subtraction operator are explained in Section 9.

42      Constant expected

A constant operand was expected, but an operand or expression that does not evaluate to a constant was supplied.

43      Operand must have segment

The SEG operator was used incorrectly. For instance, a constant operand cannot have a segment. See Section 9 for a description of valid uses of the SEG operator.

44      Must be associated with data

A code related item was used where a data related item was expected.

45      Must be associated with code

A data related item was used where a code related item was expected.

46      Multiple base registers

More than one base register was used in an operand. For example, the following line is illegal:

```
mov ax,[bx+bp]
```

47      Multiple index registers

More than one index register was used in an operand. For example, the following line is illegal:

```
mov ax,[si+di]
```

48      Must be index or base register

An indirect memory operand requires a base or index register, but some other register was specified. For example, the following line is illegal:

```
mov ax,[bx+ax]
```

Only BP, BX, DI, and SI may be used in indirect operands (except with 32 bit registers on the 80386).

49      Illegal use of register

A register was used in an illegal context. For example, the following statement is illegal:

```
mov ax,cs:si
```

50      Value out of range

A value was too large for its context. For example,

```
mov al,5000
```

is illegal; you must use a byte value for a byte register.

51      Operand not in current CS ASSUME segment

An operand was used to represent a code address outside the code segment assigned with the ASSUME statement. This usually indicates a call or jump to a label outside the current code segment.

52      Improper operand type: *symbol*

An illegal operand was given for a particular context. For example

```
mov mem1,mem2
```

is illegal if both operands are memory operands.

53      Jump out of range by *number* bytes

A conditional jump was not within the required range. For all except the 80386 processor, the range is 128 bytes backward or 127 bytes forward from the start of the instruction following the jump instruction. For the 80386, the default range is from -32,768 to 32,767. You can usually correct the problem by reversing the condition of the conditional jump and using an unconditional jump (JMP) to the out of range label, as described in Section 9.

## 55      Illegal register value

A register was specified with an illegal syntax. For example, you cannot access a stack variable with the following:

```
mov ax,bp+4
```

The correct syntax (as explained in Section 17) is shown below:

```
mov ax,[bp+4]
```

## 56      Immediate mode illegal

An immediate operand was supplied to an instruction that cannot use immediate data. For example, the following statement is illegal:

```
mov ds,DGROUP
```

You must move the segment address into a general register and then move it from that register to DS

### 57 Illegal size for operand

The size of an operand is illegal with the specified instruction. For instance, you cannot use a shift or rotate instruction with a doubleword (except on the 80386). Since this is a warning rather than an error, MASM does assemble code for the instruction, making a reasonable guess at your intention. For example, if the statement

```
inc mem32
```

is given where mem32 is a doubleword memory operand, MASM actually only increments the low order word of the operand, since a word is the largest operand that can be incremented (except on the 80386). This error may occur if you try to assemble source code written for assemblers that have less strict type checking than the CTOS Microsoft Macro Assembler. Usually you can solve the problem by specifying the size of the item with the PTR operator, as explained in Section 9.

### 58 Byte register illegal

A byte register was used in a context where a word register (or 32 bit register on the 80386) is required. For example, push al is illegal; use push ax instead.

### 59 Illegal use of CS register

The CS register was used in an illegal context, such as those listed below:

```
pop cs
mov cs,ax
```

### 60 Must be accumulator register

A register other than AL, AX, or EAX was supplied in a context where only the accumulator register is acceptable. For instance, the IN instruction requires the accumulator register as its left (destination) operand.

61            Improper use of segment register

A segment register was used in a context where it is illegal. For example, `inc cs` is illegal.

62            Missing or unreachable code segment

A jump was attempted to a label in a segment that MASM does not recognize as a code segment. This usually indicates that there is no `ASSUME` statement associating the CS register with a segment.

63            Operand combination illegal

Two operands were used with an instruction that does not allow the specified combination of operands. For example, the following operand combination is illegal:

```
xchg mem1,mem2
```

64            Near JMP/CALL to different code segment

A near jump or call instruction attempted to access an address in a code segment other than the one used in the currently active `ASSUME`. To correct the error, use a far call or jump, or use an `ASSUME` statement to change the code segment currently referenced by CS. See Section 5 for information on the `ASSUME` directive.

65            Label cannot have segment override

A segment override was used incorrectly. See Section 9 for examples of valid uses of the segment override operator.

66            Must have instruction after prefix

A repeat prefix such as `REP`, `REPE`, or `REPNE` was given without specifying the instruction to repeat.

67            Cannot override ES for destination

A segment override was used on the destination of a string instruction. Although the default DS:SI register pair for the source can have a segment override, the destination must always be in the ES:DI register pair. The ES segment cannot be overridden. For example, the following statement is illegal:

```
rep stos ds:destin ; Can't override ES
```

68            Cannot address with segment register

A statement tried to access a memory operand, but no ASSUME directive had been used to specify a segment for the operand. See Section 5 for information on the ASSUME directive.

69            Must be in segment block

A directive (such as EVEN) that is expected to be in a segment is used outside a segment.

70            Cannot use EVEN or ALIGN with byte alignment

The EVEN or ALIGN directive was used in a segment that is byte aligned. Section 6 explains the EVEN and ALIGN directives.

71            Forward reference needs override or FAR

A call or jump attempts to access a far label that was not declared far earlier in the source code. You can use the PTR operator to specify far calls and jumps, as shown below:

```
call FAR PTR task
jmp FAR PTR location
```

72            Illegal value for DUP count

The count value specified for a DUP operator did not evaluate to a constant integer greater than 0.

73 Symbol is already external

A symbol that had already been declared external was later defined locally. Section 8 describes external declarations.

74 DUP nesting too deep

DUP operators were nested to more than 17 levels.

75 Illegal use of undefined operand (?)

The undefined operand (?) was used incorrectly. For example, the following statements are illegal:

```
stuff DB 5 DUP (?+5) ; Can't use in expression
mov ax,? ; Can't use in code
```

Valid uses of the undefined operand are explained in Section 6.

76 Too many values for structure or record initialization

Too many initial values were given when declaring a record or structure variable. The number of values in the declaration must match the number in the definition. For example, a structure test defined with four fields could be declared as shown below:

```
stest test <4,, 'c',0>
```

The declaration must have four or fewer fields.

77 Angle brackets required around initialized list

A structure variable was defined without angle brackets around the initial values in the list. For example, the following definition is illegal:

```
stest test 4,, 'c'0
```

The following definitions are correct:

```
stest test <4,, 'c',0> ; Three initial values, one
 ; blank
ttest test <> ; No initial values
```



78 Directive illegal in structure

A statement within a structure definition was not one of the following: a data definition using define directives such as DB or DW, a comment preceded by a semicolon, or a conditional assembly directive.

79 Override with DUP illegal

The DUP operator was used in a structure initialization list. For example, the following example is illegal because of the DUP operator:

```
stest test <3,4 DUP (3),5>
```

80 Field cannot be overridden

An item in a structure initialization list attempted to override a structure field that could not be overridden. For instance, if a field is initialized in the structure definition with the DUP operator, it cannot be overridden in a declaration. See the note in Section 7.

83 Circular chain of EQU aliases

An alias declared with the EQU directive points to itself. For example, the following lines are illegal:

```
a EQU b
b EQU a
```

84 Cannot emulate coprocessor opcode

Either a coprocessor instruction or operands used with such an instruction produced an opcode that the coprocessor emulator does not support. Since the emulator library is not supplied with the CTOS Microsoft Macro Assembler, this error can only occur if you are linking assembler routines with code from a high level language compiler that uses the emulator.

85 End of file, no END directive

The source code was not terminated by an END statement. This error can also occur as the result of segment nesting errors.

- 86           Data emitted with no segment
- A statement that generates code or data was used outside all segment blocks. Instructions and data declarations must be in segments, but directives that specify assembler behavior without generating code or data can be outside segments.
- 87           Forced error – pass1
- An error was forced with the .ERR1 directive.
- 88           Forced error – pass2
- An error was forced with the .ERR2 directive.
- 89           Forced error
- An error was forced with the .ERR directive.
- 90           Forced error – expression true (0)
- An error was forced with the .ERRE directive.
- 91           Forced error – expression false (not 0)
- An error was forced with the .ERRNZ directive.
- 92           Forced error – symbol not defined
- An error was forced with the .ERRNDEF directive.
- 93           Forced error – symbol defined
- An error was forced with the .ERRDEF directive.
- 94           Forced error – string blank
- An error was forced with the .ERRB directive.
- 95           Forced error – string not blank
- An error was forced with the .ERRNB directive.

96 Forced error – strings identical

An error was forced with the `.ERRIDN` directive.

97 Forced error – strings different

An error was forced with the `.ERRDIF` directive.

98 Wrong length for override value

The override value for a structure field is too large to fit in the field. An example is shown below:

```
x STRUC
x1 DB "A"
x ENDS
y x < "AB" >
```

The override value is a string consisting of two bytes; the structure declaration provided only room for one byte.

99 Line too long expanding symbol: *symbol*

An equate defined with the `EQU` directive was so long that expanding it caused the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

100 Impure memory reference

Data was stored into the code segment when the `/P` option and privileged instructions (enabled with `.286P` or `.386P`) were in effect. An example of storing data in the code segment is shown below:

```
 .CODE
c_word DW ?
 .
 .
 .
 mov cs:c_word,data
```

The `/P` option checks for such statements, which are acceptable in real mode, but can cause problems in privileged mode.

## 101 Missing data; zero assumed

An operand is missing from a statement, as shown below:

```
mov ax,
```

Since some programmers use this syntax purposely, the message is a warning. MASM assumes that 0 was intended and assembles the following code:

```
mov ax,0
```

## 102 Segment near (or at) 64K limit

A bug in the 80286 processor causes jump errors when a code segment approaches within a few bytes of the 64K limit in privileged mode. This error warns about code that may fail because of the bug. The error can only be generated when the .286 directive is given.

## 103 Align must be power of 2

A number that is not a power of two was used with the ALIGN directive. The directive is explained in Section 6.

### 104      Jump within short distance

A JMP instruction was used to jump to a short label (128 or fewer bytes before the end of the JMP instruction, or 127 or fewer bytes beyond the instruction). By default the assembler assumes that jumps are near (greater than short, but still in one segment). If a short jump is encountered, MASM uses a short form of the JMP instruction (2 bytes) rather than the long form (3 bytes with 16 bit segments or 5 bytes with 32 bit segments). You can make your code slightly more efficient by using the SHORT operator to specify that a jump is short rather than near. For example, using the SHORT operator in the following example saves 1 byte of code:

```
 jmp SHORT there
 .
there: . ; Less than 127 bytes
```

Using the SHORT operator with forward references to code labels is explained in Section 9. With the 80386 processor, this message also applies to conditional jumps, which can be either short (2 bytes) or near (4 bytes).

### 105      Expected *element*

An element such as a punctuation mark or operator was omitted. For instance, if you omit the comma between source and destination operands, the message Expected comma is generated.

### 106      Line too long

A source line was longer than 128 characters, the maximum allowed by MASM.

### 107      Illegal digit in number

A constant number contained a digit that is not allowed in the current radix.

108      Empty string not allowed

A statement used an empty string. For example, the following definition is illegal:

```
 null DB
```

In many languages an empty string represents ASCII character 0. In assembly language, you must give the value 0, as shown below:

```
 null DB 0
```

109      Missing operand

The instruction or directive requires more operands than were provided.

110      Open parenthesis or bracket

Only one parenthesis or bracket was given in a statement that requires opening and closing parentheses or brackets.

111      Directive must be in macro

A directive that is expected only in macro definitions was used outside a macro.

112      Unexpected end of line

A line ended before a complete statement was formed. MASM expects more information, but cannot identify what information is missing.

113      Cannot change processor in segment

A processor directive was encountered within a segment. Processor directives must be given before the first segment directive or between segments. If you want to change the processor in the middle of the segment, you must close the current segment, give the processor directive, and then start another segment.

### 114 Operand size does not match segment word size

A 32 bit operand was used in a 16 bit segment, or vice versa. This warning can only occur with the 80386. For example, the following statement is a questionable practice in a 32 bit segment:

```
mov ax,OFFSET nearlabel ; Load near (32 bit)
 ; label
```

The following statement is a questionable practice in a 16 bit segment:

```
mov eax,OFFSET farlabel ; Load far (48 bit)
 ; label
```

This is a warning that you can ignore if you are certain you know what you are doing.

### 115 Address size does not match segment word size

A 32 bit address was used in a 16 bit segment, or vice versa. This warning can only occur with the 80386. For example, the following statement is a questionable practice in a 32 bit segment:

```
mov eax,[si] ; Load value pointed to by 16 bit
 ; pointer
```

The following statement is a questionable practice in a 16 bit segment:

```
mov ax,[esi] ; Load value pointed to by 32 bit
 ; pointer
```

This is a warning that you can ignore if you are certain you know what you are doing.

### 116 Include file not found: *filename*

The include file specified was not found. Check to see if the file is in the directory specified in the environment. Also check to see if the full pathname is required and is correct.

## Unnumbered Error Messages

Unnumbered messages appear when an error occurs that cannot be associated with a particular line of code. Generally these errors indicate problems with the command line, memory allocation, or file access. MASM may generate the unnumbered error messages below.

### File-Access Errors

Any of the following errors may occur when MASM tries to access a file for processing. They usually indicate insufficient disk space, a corrupted file, or some other file error.

End of file encountered on input file

Read error on standard input

Unable to access input file: *filename*

Unable to open cref file: *filename*

Unable to open input file: *filename*

Unable to open listing file: *filename*

Unable to open object file: *filename*

Write error on cross-reference file

Write error on listing file

Write error on object file

### Command-Line Errors

Any of the following errors may occur if you give an invalid command line when starting MASM.

Error defining symbol “*name*” from command line

Warning level (0-2) expected after W option

Path expected after I option

Source file expected; not found

Extra file name ignored

Line invalid, start again



Path expected after I option

Unknown case option: *option*. Use /help for list of options.

Unknown option: *option*. Use /help for list of options.

### Miscellaneous Errors

The following errors indicate a problem with memory allocation or some other assembler problem that is not related to a specific source line.

Internal error

Note the conditions when the error occurs and report them to Unisys Corporation.

Internal unknown error

This error may indicate that the internal error table has been corrupted and MASM cannot figure out what the error is. Note the conditions when the error occurs and report them to Unisys Corporation. The following errors indicate a problem with memory allocation or some other assembler problem not related to a specific source line.

Number of open conditionals: < number >

Conditional assembly directives (starting with IF) were given without corresponding ENDIF directives.

Open procedures

A PROC directive was given without a corresponding ENDP directive.

Open segments

A segment was defined, but never terminated with an ENDS directive. This error does not occur with simplified segment directives.

Out of memory

All available memory has been used, either because the source file is too long, or because there are too many symbols defined in the symbol table.

You can solve this problem in several ways. First, try assembling with no listing or cross reference file. If this works, you can reassemble by specifying a null object file to get a listing or cross reference file. You can also rewrite the source file to require less symbol space. Techniques for reducing symbol space include minimizing use of macros, equates, and structures; using short symbol names; using tab characters in macros rather than series of spaces; using macro comments (;;) rather than normal comments (;); and purging macro definitions after last use.

## MASM Exit Codes

The assembler returns one of the following codes after an assembly. The codes can be tested by a make file or batch file.

| <b>Code</b> | <b>Meaning</b> |
|-------------|----------------|
|-------------|----------------|

|    |                                              |
|----|----------------------------------------------|
| 0  | No error                                     |
| 1  | Argument error                               |
| 2  | Unable to open input file                    |
| 3  | Unable to open listing file                  |
| 4  | Unable to open object file                   |
| 5  | Unable to open cross-reference file          |
| 6  | Unable to open include file                  |
| 7  | Assembly error                               |
| 8  | Memory-allocation error                      |
| 9. | Real number input not allowed                |
| 10 | Error defining symbol from command line (/D) |
| 11 | Assembly interrupted                         |

Note that if the exit code is 7, MASM automatically deletes the invalid object file.

## CREF Error Messages and Exit Codes

The CTOS Microsoft Cross Reference Utility, CREF, terminates operation and displays one of the following messages when it encounters an error:

Can't open cross reference file for reading

The cross reference file was not found. Make sure the file is on the specified disk and that the name is spelled correctly in the command line.

Can't open listing file for writing

The disk is full or write protected, a file with the specified name already exists, or the specified volume is not available.

CREF has no options

An option was specified in the command line with the slash (/) or dash (-) character, but CREF has no options.

Extra file name ignored

More than two files were specified on the command line. CREF uses only the first two files given.

Line invalid, start again

No cross reference file was provided in the command line or at the prompt. CREF displays this message followed by a prompt asking for the file.

Out of heap space

CREF did not find enough memory to process the files. Try again with no resident programs or DOS shells, or add more memory.

Premature EOF

The file specified was not a valid cross reference file, or the file was damaged.

Read error on standard input

An input error was received from the command line or submit file.

CREF only returns two exit codes: 0 if the program is successful, or 1 if an error occurs.

B- PRELIMINARY 4164 0574-000

# Glossary

## A

### **ASCII.**

The American Standard Code for Information Interchange (ASCII) defines the character set codes used for information exchange between equipment.

### **.Asm.**

.Asm is the standard file name suffix for assembly language source code files.

### **Assembler.**

The Assembler translates Assembly 8086 programs into CTOS object modules (machine code).

### **Assembly Language Statements.**

An assembly language statement is a combination of mnemonics, operands, and comments that defines the object code to be created at assembly time.

## B

### **Bind.**

Bind is a command that activates the Linker to create a version 6 run file. Version 6 run files are required for protected mode compatibility.

## C

### **Case Sensitivity.**

Case sensitivity means that upper and lower case letters are considered different. If case sensitivity is turned on then names that have the same spelling but use letters of different cases are considered different.

### **Code segment.**

A code segment is a variable-length (up to 64K) logical entity consisting of re-entrant code, and containing one or more complete procedures.

**Command Line.**

A command line is input entered at the executive prompt to invoke a program.

**Comment.**

A comment is text which the programmer includes for documentary purposes only. It is ignored by the assembler.

**Compiler.**

A compiler is a program which translate high level language source code programs into object modules.

**Conditional Assembly.**

Conditional assembly allows the programmer to test conditions at assembly time and selectively assemble statements.

**Constant.**

A constant is a value which is initialized at assembly time and which does not change during program execution.

**CREF.**

See Cross Reference Utility.

**Cross Reference Utility.**

The Cross Reference Utility (CREF) is a program which converts a binary cross reference file, produced by the assembler, into a readable ASCII file.

**CTOS.Lib.**

The CTOS.Lib is part of the Language Development software. It is a library of object modules that provide operating system run-time support.

## D

**Delimiter.**

A character (or sequence of contiguous characters) that separates a string of characters from another string of characters.

**DGroup.**

DGroup usually includes data, constant, and stack Linker segments.

**Directive.**

A directive tells the assembler how to generate code and data at assembly time. A directive does not in itself generate any code.

## E

### **8086 Assembly Language.**

8086 Assembly language is the low level language you can use to write programs. You use MASM to convert the programs into object modules.

### **Expression.**

In a program, an expression is a combination of various constants, variables, operators, and parentheses used to perform a desired computation.

### **External reference.**

An external reference is a reference from one object module to a variable in another object module.

## G

### **Group.**

A group is a named collection of Linker segments that the CTOS loader addresses at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

## J

### **Jump.**

Transfer of control from one portion of a program to another.

## L

### **Label.**

A label is a symbolic name for the address of an instruction to which control can be transferred using a jump instruction.

### **.Lib.**

.Lib is the standard file name suffix for library files.

### **Librarian.**

The Librarian is a program that creates and maintains object module libraries. The Linker can search automatically in such libraries to select only those object modules that a program calls.

**Library.**

A library is a stored collection of object modules (complete routines or subroutines) that are available for linking into run files.

**Library file.**

A library file can contain one or more object modules. The file name normally includes the suffix `.Lib`.

**Link.**

Link is the Executive command that activates the linker to create Version 4 run files. Version 4 run files are not protected mode compatible.

**Linker.**

The Linker is a program that combines object modules (files that compilers and assemblers produce) into run files.

**Linker segment.**

A Linker segment is a single entity consisting of all segment elements with the same segment name.

**.Lst.**

`.Lst` is the standard file name suffix for listing files produced by the assembler.

## M

**Macro.**

A macro (short for macroinstruction) is a single instruction that represents a given sequence of instructions. The macro is defined to represent a set of instructions and can be used each time to represent that set.

**.Map.**

`.Map` is the standard file name suffix for Linker list files.

**Memory Model**

The memory model specifies the default size of data and code used in a program.

**Model.**

See Memory Model.

## N

### **NCP.**

See Numeric Co-Processor.

### **Numeric Co-Processor.**

An Intel 8087, 80x87, or 80387. When present, it is used to speed up the processing of numerical calculations.

## O

### **.Obj.**

.Obj is the standard file name suffix for object module files.

### **Object Code.**

Object code is machine code which output from an assembler or compiler.

### **Object File.**

See Object Module.

### **Object Module.**

An object module is the result of a single compiler or assembler function. You can link the object module with other object modules into run files.

### **Op Code.**

An op code is the operation code which is generated from an assembly instruction by the assembler. It is recognized and executed by the processor at run time.

### **Operand.**

An operand is the target register or address on which an operation code operates.

## P

### **Parameter.**

A parameter is a variable or constant that is transferred to and from a subroutine or program.

### **Physical address.**

A physical address does not specify a segment base and is relative to memory location 0.

### **Pointer.**

A pointer is an address that specifies a storage location for data.



**Public procedure.**

A public procedure is a procedure which can be referenced by a module other than the defining module.

**Public variable.**

A public variable is a variable which can be referenced by a module other than the defining module.

## R

**Radix.**

A radix is the base of a number such as binary, octal, decimal or hexadecimal.

**Register.**

Registers are special areas of memory. They are internal to the processor and therefore accessed faster than regular memory.

**Repeat Block.**

A repeat block is a special construct supported by the 8086 family of processors which allows a series of instructions to be repeated conditionally.

**Resident.**

The resident portion of a program remains in memory throughout execution.

**.Run.**

.Run is the standard file name suffix for run files.

**Run file.**

A run file is a complete program: a memory image of a task in relocatable form, linked into the standard format CTOS requires. You use the Linker to create run files.

**Run-file checksum.**

The run-file checksum is a number the Linker produces based on the summation of words in the file. The system uses the checksum to check the validity of the run file.

## S

### **Segment.**

A segment is a contiguous area of memory that consists of an integral number of paragraphs. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind can be either shared or nonshared.

### **Segment address.**

The segment address is the segment base address. For an 8086/80186 microprocessor, a segment address refers to a paragraph (16 bytes).

### **Segmented address.**

A segmented address is an address that specifies both a segment base and an offset.

### **Segment override.**

Segment override is operating code that causes the 8086/80186 to use the segment register specified by the prefix instead of the segment register that it would normally use when executing an instruction.

### **Segment registers.**

All addresses are relative to one of the following segment registers: the Code Segment (CS) register, the Data Segment (DS) register, the Extra Segment (ES) register, and the Stack Segment (SS) register.

### **Simplified Segments.**

Simplified segments are segment definitions which use a simple keyword to define a series of default segments which are most commonly used.

### **Source file.**

A source file is user readable text which represents a program to be assembled or compiled into an object file.

### **Stack.**

A stack is a region of memory accessible in LIFO (Last in first out) order. It is commonly used to temporarily store parameters passed from one routine another.

### **Stack frame.**

The stack frame is a region of a stack corresponding to the dynamic invocation of a procedure. It consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

**Stack pointer.**

A stack pointer (SP) indicates the top of a stack. The stack pointer is stored in the registers SS:SP.

**Statements.**

See Assembly Language Statements.

**Strings.**

A string is a sequence of bytes, usually text characters, which can be manipulated using specialized string handling instructions.

**.Sym.**

.Sym is the standard file name suffix for the symbol file.

**Symbol file.**

The Linker symbol file (suffix .Sym) contains a list of all public symbols.

**Symbol Table.**

A symbol table is a list of symbols defined in a program. It is output by the assembler along with the assembly listing.

**Symbolic instructions.**

Symbolic instructions are instructions containing mnemonic characters corresponding to assembly language instructions. These instructions cannot contain user-defined public symbols.

## T

**Text Editor.**

A text editor is an interactive program which allows a user to create text files.

**Text file.**

A text file is a file which contains a sequence of mostly printable characters but may contain some controls such as tabs, line feeds, and page ejects. A text file usually contains readable text such as documents or source programs.

# Index

- .186 directive, 6–13
- .286 directive, 6–13
- .286P directive, 6–14
- .287 directive, 6–14
- .386, 6–14
- .386P directive, 6–14
- .387 directive, 6–15
- 80386 processor, 15–15, 22–4
  - instructions, A–1
- .8086 directive, 6–13
- 8086 family processors, 15–1
  - differences, 15–2
- .8087 directive, 6–14

## A

- AAA instruction, 18–12
- AAD instruction, 18–13
- AAM instruction, 18–13
- AAS instruction, 18–12
- ADC instruction, 18–1
- ADD instruction, 18–1
- Adding, 18–1
  - multiple registers, 18–3
- Addressing
  - immediate, 16–2
  - memory, 16–4
  - memory, direct, 16–4
  - memory, indirect, 16–6
  - modes, 16–1
  - register, 16–3
  - segmented, 15–4
- ALIGN, 8–23

- ALIGN directive, 8–23
- Aligning data, 8–23
- .ALPHA directive, 7–14
- AND instruction, 18–16, 19–4
- Arguments
  - passing on the stack, 19–19
- Arithmetic, 18–1
  - adding, 18–1
  - BCD, 18–11
  - dividing, 18–9
  - multiplying, 18–7
  - subtracting, 18–4
- Arithmetic operators, 11–4
- Arrays and buffers, 8–20
- Assembling conditionally, 12–1
- Assembly language
  - statements, 6–1
- Assembly statistics, 4–28
- ASSUME directive, 7–26, 11–11

## B

- Binary coded decimal, 18–11
  - packed, 18–14
  - unpacked, 18–12
- Bit manipulations, 18–1
  - AND, 18–16
  - logical, 18–15
  - NOT, 18–18
  - OR, 18–17
  - rotating, 18–20
  - scanning, 18–19
  - shifting, 18–20
  - XOR, 18–17

- Bitwise logical operators, 11–8
- BSF instruction, 18–19
- BSR instruction, 18–19
- BT instruction, 19–11
- BTC instruction, 19–11
- BTR instruction, 19–11
- BTS instruction, 19–12

## C

- Calculation operators, 11–4
- CALL instruction, 17–10, 19–16
- Case sensitivity, 4–14
- CBW instruction, 17–5
- CDQ instruction, 17–6
- CLC instruction, 18–4
- .CLD instruction, 20–2
- CLI instruction, 19–27
- CMP instruction, 19–4
- CMPS instructions, 20–9
- .CODE directive, 7–5
- COMM, 10–9, 10–10
- Command line help, 4–13
- COMMENT directive, 6–4
- Comments, 6–3
  - macro, 13–19
- Communal symbols, 10–9
- Conditional assembly, 12–1
  - directives, 12–1
  - error directives, 12–6
- Conditional jump, 19–4
- Conditional jump
  - summary, 19–6
- Configuration Strategy, 3–1
- .CONST directive, 7–5
- Constants, 6–7
  - integer, 6–7
  - real number, 6–10
  - strings, 6–11

- Controlling assembly output, 14–1
  - cross reference, 14–8
  - listings, 14–5
- Controlling program flow, 19–1
  - jumping, 19–1
  - looping, 19–12
  - procedures, 19–15
  - SET, 19–14
- Converting data, 17–5
- Coprocessor, 4–12
- CREF, 5–1
  - command line, 5–2
  - cross reference listing, 5–2, 5–3
  - error messages, B–26
  - prompts, 5–3
  - summary, 3–8
- .CREF directive, 14–8
- Cross reference files
  - listing, 4–21
  - output, 4–2
  - specifying, 4–14
- Cross reference utility, 1–2
- CWD instruction, 17–5
- CWDE instruction, 17–6

## D

- DAA instruction, 18–14
- DAS instruction, 18–14
- Data
  - converting, 17–5
  - copying, 17–2
  - exchanging, 17–3
  - initializing, 8–6
  - loading pointers, 17–7
  - looking up, 17–3
  - moving, 17–1
  - ports, 17–15
  - stack, 17–10
  - transferring flags, 17–4
- DB directive, 8–7

- .DATA directive, 7-5
- .DATA? directive, 7-5
- DD directive, 8-7
- Debugging, 3-8
- DEC instruction, 18-4
- Default
  - assembly behavior, 6-12
  - radix, 6-8, 6-9
- Defining symbols
  - from command line, 4-11
- Development Cycle, 3-2
- DF directive, 8-7
- Directives
  - .186, 6-13
  - .286, 6-13
  - .286P, 6-14
  - .287, 6-14
  - .386, 6-14
  - .386P, 6-14
  - .387, 6-15
  - .8086, 6-13
  - .8087, 6-14
  - ALIGN, 8-23
  - .ALPHA, 7-14
  - ASSUME, 7-26, 11-11
  - .CODE, 7-5
  - COMM, 10-10
  - COMMENT, 6-4
  - conditional assembly, 12-1
  - .CONST, 7-5
  - .CREF, 14-8
  - .DATA, 7-5
  - .DATA?, 7-5
  - DB, 8-7
  - DD, 8-7
  - DF, 8-7
  - DOSSEG, 7-14
  - DQ, 8-7
  - DT, 8-7
  - DW, 8-7
  - ELSE, 12-2
- Directives (continued)
  - END, 6-16
  - ENDIF, 12-2
  - ENDM, 13-6
  - ENDP, 8-4, 19-16
  - ENDS, 7-16, 9-2
  - EQU, 10-4, 13-2
  - equal sign (=), 13-2
  - .ERR, 12-7
  - .ERR1, 12-7
  - .ERR2, 12-7
  - .ERRB, 12-10
  - .ERRDEF, 12-9
  - .ERRDIF, 12-10
  - .ERRE, 12-8
  - .ERRIDN, 12-10
  - .ERRNB, 12-10
  - .ERRNDEF, 12-9
  - .ERRNZ, 12-8
  - EVEN, 8-23
  - EXITM, 13-10
  - EXTRN, 10-4
  - .FARDATA, 7-5
  - .FARDATA?, 7-5
  - GROUP, 7-26, 11-11
  - IF, 12-2, 12-3
  - IF1, 12-3
  - IF2, 12-3
  - IFB, 12-4
  - IFDEF, 12-4
  - IFDIF, 12-5
  - IFE, 12-3
  - IFIDN, 12-5
  - IFNB, 12-4
  - IFNDEF, 12-4
  - INCLUDE, 13-24
  - IRP, 13-12
  - IRPC, 13-13
  - LABEL, 8-3, 8-5
  - .LALL, 13-8, 14-7
  - .LFCOND, 14-6

### Directives (continued)

- .LIST, 14-5
- LOCAL, 13-9
- MACRO, 13-6
- .MODEL, 7-4
- .MSFLOAT, 6-11
- NAME, 10-9
- operands, 11-2
- %OUT, 14-1
- PAGE, 14-3
- PROC, 7-10, 8-4, 19-16, 19-17
- PUBLIC, 10-3
- PURGE, 13-25
- .RADIX, 6-8
- RECORD, 9-6
- REPT, 13-12
- .SALL, 13-8, 14-7
- SEGMENT, 7-16, 11-11
- .SEQ, 7-15
- .SFCOND, 14-6
- .STACK, 7-5
- STRUCT, 9-2
- SUBTTL, 14-3
- summary, 6-13
- .TFCOND, 14-6
- TITLE, 14-2
- .XALL, 13-8, 14-7
- .XCREF, 14-8
- .XLIST, 14-5
- DIV instruction, 18-9
- Dividing, 18-9
- DOSSEG directive, 7-14
- DQ directive, 8-7
- DS register, 7-32
- DT directive, 8-7
- DW directive, 8-7

## E

- ELSE directive, 12-2
- Emulator, 4-12

- END, 6-16
- ENDIF directive, 12-2
- ENDM directive, 13-6
- ENDP directive, 8-4, 19-16
- ENDS directive, 7-16, 9-2
- ENTER instruction, 19-24
- Environment Variables, 3-2
- Environment variables, 4-5
- EQU directive, 10-4, 13-2
- Equal sign (=) directive, 13-2
- Equates, 13-2
  - managing, 13-24
  - nonredefinable numeric, 13-3
  - redefinable numeric, 13-2
  - string, 13-4
- .ERR directive, 12-7
- .ERR1 directive, 12-7
- .ERR2 directive, 12-7
- .ERRB directive, 12-10
- .ERRDEF directive, 12-9
- .ERRDIF directive, 12-10
- .ERRE directive, 12-8
- .ERRIDN directive, 12-10
- .ERRNB directive, 12-10
- .ERRNDEF directive, 12-9
- .ERRNZ directive, 12-8
- Error lines, 4-18
- Error messages, B-1
  - CREF, B-26
  - MASM, B-1
  - miscellaneous, B-24
  - numbered assembler, B-2
  - unnumbered, B-23
- ESC instruction, 22-2
- EVEN, 8-23
- EVEN directive, 8-23
- Exit codes, B-1, B-25
- EXITM directive, 13-10
- Expressions, 11-1
- External symbols, 10-4
- EXTRN, 10-4

**F**

False conditionals, 4–18  
.FARDATA directive, 7–5  
.FARDATA? directive, 7–5  
File buffer size, 4–10  
Forward references, 11–22  
    labels, 11–22  
    variables, 11–25

**G**

General purpose registers, 15–9  
Group  
    table, 4–25  
GROUP directive, 7–26, 11–11

**H**

HIGH and LOW operators, 11–13  
HLT instruction, 22–2

**I**

IDIV instruction, 18–9  
IF directive, 12–2, 12–3  
IF1 directive, 12–3  
IF2 directive, 12–3  
IFB directive, 12–4  
IFDEF directive, 12–4  
IFDIF directive, 12–5  
IFE directive, 12–3  
IFIDN directive, 12–5  
IFNB directive, 12–4  
IFNDEF directive, 12–4  
Immediate operands, 16–2  
Impure code, checking for, 4–15  
IMUL instruction, 18–7  
IN instruction, 17–15  
INC instruction, 18–1  
INCLUDE directive, 13–24

**Include Files**

    Setting a search path, 4–13  
Index operator, 11–6  
Initializing data, 8–6  
    arrays and buffers, 8–20  
    pointer variables, 8–12  
    real number variables, 8–14  
    segment registers, 7–30  
    string variables, 8–12  
    variables, 8–6

INS instructions, 20–13

Install, 2–3

Installation, 2–1

**Instructions**

AAA, 18–12  
AAD, 18–13  
AAM, 18–13  
AAS, 18–12  
ADC, 18–1  
ADD, 18–1  
AND, 18–16, 19–4  
BSF, 18–19  
BSR, 18–19  
BT, 19–11  
BTC, 19–11  
BTR, 19–11  
BTS, 19–12  
CALL, 17–10, 19–16  
CBW, 17–5  
CDQ, 17–6  
CLC, 18–4  
CLD, 20–2  
CLI, 19–27  
CMP, 19–4  
CMPS, 20–9  
    coprocessor, 21–1  
CWD, 17–5  
CWDE, 17–6  
DAA, 18–14  
DAS, 18–14  
DEC, 18–4



### Instructions (continued)

DIV, 18-9  
ENTER, 19-24  
ESC, 22-2  
HLT, 22-2  
IDIV, 18-9  
IMUL, 18-7  
IN, 17-15  
INC, 18-1  
INS, 20-13  
INT, 17-10, 19-25  
INTO, 19-25  
IRET, 17-10, 19-27  
JC, 18-2  
Jcondition, 19-4, 19-5, 19-8  
JCXZ, 19-13  
JECXZ, 19-13  
JMP, 19-2  
LAHF, 17-4  
LDS, 17-9  
LEA, 17-8  
LEAVE, 19-24  
LES, 17-9  
LFS, 17-9  
LGS, 17-9  
LOCK, 22-2  
LODS, 20-11  
LOOP, 19-12  
LOOPE, 19-13  
LOOPNE, 19-13  
LOOPNZ, 19-13  
LOOPZ, 19-13  
LSS, 17-9  
MOV, 17-2  
MOVS, 20-5  
MOVSX, 17-6  
MOVZX, 17-6  
MUL, 18-7  
NEG, 18-4  
NOP, 22-1  
NOT, 18-18

### Instructions (continued)

OR, 18-17  
OUT, 17-15  
OUTS, 20-13  
POP, 17-10  
POPA, 17-15  
POPAD, 17-15  
POPD, 17-15  
POPF, 17-14  
POPFD, 17-14  
PUSH, 17-10  
PUSHA, 17-15  
PUSHAD, 17-15  
PUSHD, 17-15  
PUSHF, 17-14  
PUSHFD, 17-14  
RCL, 18-20  
RCR, 18-20  
REP, 20-3  
REPE, 20-3  
REPNE, 20-3  
REPNZ, 20-3  
REPZ, 20-3  
RET, 17-10, 19-16, 19-17  
RETF, 19-17  
RETN, 19-17  
ROL, 18-20  
ROR, 18-20  
SAHF, 17-4  
SAL, 18-20  
SAR, 18-20  
SBB, 18-4  
SCAS, 20-7  
SET, 19-14  
SHL, 18-20  
SHLD, 18-25  
SHR, 18-20  
SHRD, 18-25  
STI, 19-27, 20-2  
STOS, 20-10  
SUB, 18-4

## Instructions (continued)

- TEST, 19-9
- WAIT, 21-11, 22-2
- XCHG, 17-3
- XLAT, 17-3
- XOR, 18-17
- INT instruction, 17-10, 19-25
- Integer constants, 6-7
- Integer variables, 8-7
- Interrupts, 19-25
  - calling, 19-25
- INTO instruction, 19-25
- IRET instruction, 17-10, 19-27
- IRP directive, 13-12
- IRPC directive, 13-13

**J**

- JC instruction, 18-2
- Jcondition instruction, 19-4, 19-5, 19-8
- JCXZ instruction, 19-13
- JECXZ instruction, 19-13
- JMP instruction, 19-2
- Jumping, 19-1
  - comparing, 19-4
  - conditionally, 19-4
  - flags, 19-8
  - testing bits, 19-9
  - unconditionally, 19-2

**L**

- LABEL, 8-5, 8-21
- LABEL directive, 8-3
- Labels, 8-1
  - code, 8-3
  - procedure, 8-4
- LAHF instruction, 17-4
- .LALL directive, 13-8, 14-7
- LDS instruction, 17-9

- LEA instruction, 17-8
- LEAVE instruction, 19-24
- LENGTH operator, 11-17
- LES instruction, 17-9
- .LFCOND directive, 14-6
- LFS instruction, 17-9
- LGS instruction, 17-9
- Library files, 3-8
- Linking, 3-8
- .LIST directive, 14-5
- Listing
  - abbreviations, 4-21
  - assembly statistics, 4-28
  - code, 4-20
  - cross reference, 5-3
  - described, 4-19
  - errors, 4-20
  - pass 1, 4-10, 4-28
  - suppressing tables, 4-15
- LOCAL directive, 13-9
- Local variables, 19-21
- Location counter
  - setting, 8-22
  - using, 11-21
- LOCK instruction, 22-2
- LODS instructions, 20-11
- LOOP instruction, 19-12
- LOOPE instruction, 19-13
- Looping, 19-12
- LOOPNE instruction, 19-13
- LOOPNZ instruction, 19-13
- LOOPZ instruction, 19-13
- LSS instruction, 17-9

**M**

- Macro comments, 13-19
- MACRO directive, 13-6

Macro operators, 13–14  
    expression, 13–18  
    literal character, 13–17  
    literal text, 13–16  
    substitute, 13–15

Macros, 13–5  
    calling, 13–8  
    defining, 13–6  
    exiting from, 13–10  
    local symbols, 13–9  
    managing, 13–24  
    nested, 13–21  
    purging, 13–25  
    recursive, 13–20  
    redefining, 13–23

MASK operator, 9–12

MASM  
    command line, 4–2  
    environment variables, 4–5  
    invoking, 4–1  
    options, 4–8  
    prompts, 4–4  
    summary, 3–7

Math coprocessor, 21–1  
    architecture, 21–2  
    arithmetic, 21–19  
    constants, 21–17  
    controlling, 21–33  
    emulator, 21–5  
    instructions, 21–5  
    memory access, 21–11  
    operands, 21–7, 21–8, 21–9,  
        21–10  
    program flow, 21–26  
    registers, 21–2, 21–3  
    transcendental  
        instructions, 21–31  
    transferring data, 21–12, 21–18

Memory addressing  
    direct, 16–4  
    indirect, 16–6  
    80386 indirect, 16–11

Memory models, 7–3  
    defining, 7–4

Memory operands, 16–4

Message output, 4–7

Mnemonics  
    defined, 6–3  
    reserved names, 6–6

.MODEL directive, 7–4

MOV instruction, 17–2

MOVS instructions, 20–5

MOVSB instruction, 17–6

MOVZX instruction, 17–6

.MSFLOAT directive, 6–11

MUL instruction, 18–7

Multiple modules, 10–1, 10–7

Multiplying, 18–7

## N

NAME directive, 10–9

Names  
    assigning to symbols, 6–4

NEG instruction, 18–4

New features, 1–2, A–1

NOP instruction, 22–1

NOT instruction, 18–18

## O

OFFSET operator, 11–14

Operands, 11–1  
    immediate, 16–2  
    memory, 16–4  
    register, 16–3  
    structure, 9–4

Operator precedence, 11–19

Operators, 11–3

arithmetic, 11–4

bitwise logical, 11–8

calculation, 11–4

HIGH and LOW, 11–13

index, 11–6

LENGTH, 11–17

OFFSET, 11–14

precedence, 11–19

PTR, 11–12

record, 9–12

relational, 11–9

SEG, 11–14

segment override, 11–10

shift, 11–8

SHORT, 11–12

SIZE, 11–18

structure field name, 11–6

THIS, 11–13

type, 11–11

TYPE, 11–16

.TYPE, 11–15

Options, 4–8

OR instruction, 18–17

%OUT directive, 14–1

OUT instruction, 17–15

OUTS instruction, 20–13

## P

PAGE directive, 14–3

Pass 1 listing, 4–10, 4–28

Passing arguments, 19–19

Pointers, 17–7

POP instruction, 17–10

POPA instruction, 17–15

POPAD instruction, 17–15

POPD instruction, 17–15

POPF instruction, 17–14

POPFD instruction, 17–14

Ports

strings, 20–13

Predefined equates, 7–8

PROC directive, 7–10, 8–4, 19–16,  
19–17

Procedures, 19–15

calling, 19–16

defining, 19–17

Processor

controlling, 22–1

protected mode, 22–3

80386, 22–4

PTR operator, 11–12

PUBLIC, 10–3

Public symbols, 10–3

PURGE directive, 13–25

PUSH instruction, 17–10

PUSHA instruction, 17–15

PUSHAD instruction, 17–15

PUSHD instruction, 17–15

PUSHF instruction, 17–14

PUSHFD instruction, 17–14

## R

Radix

binary, 6–8

default, 6–8, 6–9

specifiers, 6–8

.RADIX directive, 6–8

RCL instructions, 18–20

RCR instruction, 18–20

Real number constants, 6–10

RECORD directive, 9–6

Records, 9–5

operands, 9–11

operators, 9–12

table, 4–24

types, 9–6

variables, 9–8

Register operands, 16–3

Registers, 15–6  
  DS, 7–32  
  flags, 15–12  
  general purpose, 15–9  
  other, 15–11  
  segment, 7–30, 15–9  
  SS and SP, 7–33  
  8087, 15–14  
Relational operators, 11–9  
REP instruction, 20–3  
REPE instruction, 20–3  
Repeat blocks  
  defining, 13–11  
  IRP directive, 13–12  
  IRPC directive, 13–13  
  REPT directive, 13–12  
Repeat instructions, 20–2  
REPNE instruction, 20–3  
REPNZ instruction, 20–3  
REPT directive, 13–12  
REPZ instruction, 20–3  
Reserved names, 6–6  
RET instruction, 17–10, 19–16,  
  19–17  
RETF instruction, 19–17  
RETN instruction, 19–17  
ROL instruction, 18–20  
ROR instruction, 18–20

## S

SAHF instruction, 17–4  
SAL instruction, 18–20  
.SALL directive, 13–8, 14–7  
SAR instruction, 18–20  
SBB instruction, 18–4  
SCAS instructions, 20–7  
SEG operator, 11–14

Segment  
  defaults, 7–10  
  definitions, full, 7–14, 7–16  
  definitions, simplified, 7–2, 7–5  
  groups, 7–25  
  nesting, 7–34  
  registers, 7–28  
  registers, initializing, 7–30  
  structure, 7–1  
  table, 4–25  
SEGMENT directive, 7–16, 11–11  
Segment order method  
  setting, 7–14  
Segment override operators, 11–10  
Segment registers, 15–9  
Segmented addresses, 15–4  
Segment–Order method  
  summary, 4–9  
.SEQ directive, 7–15  
SET instruction, 19–14  
.SFCOND directive, 14–6  
Shift operators, 11–8  
SHL instruction, 18–20  
SHLD instruction, 18–25  
SHORT operator, 11–12  
SHR instruction, 18–20  
SHRD instruction, 18–25  
SIZE operator, 11–18  
Software Installation, 2–2  
Source code  
  END directive, 6–16  
  format, 6–1  
  samples, 6–2  
Stack, 17–10  
  local variables, 19–21  
  passing arguments, 19–19  
  stack frames, 19–24  
  using, 17–13  
.STACK directive, 7–5

Statements  
    assembly language, 6–1  
Statistics, 4–16, 4–28  
STI instruction, 19–27, 20–2  
STOS instructions, 20–10  
Strings  
    comparing, 20–9  
    constants, 6–11  
    filling, 20–10  
    loading values, 20–11  
    moving, 20–5  
    ports, 20–13  
    processing, 20–1  
    searching, 20–7  
    variables, 8–12  
STRUCT directive, 9–2  
Structure field name  
    operator, 11–6  
Structures, 9–1  
    operands, 9–4  
    table, 4–24  
    types, 9–2  
    variables, 9–3  
SUB instruction, 18–4  
Subtracting, 18–4  
    multiple registers, 18–6  
SUBTTTL directive, 14–3  
Symbol table, 4–26  
Symbolic information, 4–18  
Symbols, 4–11, 6–4  
    communal, 10–9  
    external, 10–4  
    public, 10–3

## T

TEST instruction, 19–9  
.TFCOND directive, 14–6  
THIS operator, 11–13

TITLE directive, 14–2  
TYPE operator, 11–16  
.TYPE operator, 11–15  
Type operators, 11–11  
Type specifiers, 8–1  
    summary, 8–2

## U

Unconditional jump, 19–2

## V

Variables, 8–1, 8–6  
    integer, 8–7  
    labels, 8–21  
    local, 19–21  
    pointer, 8–12  
    real number, 8–14  
    record, 9–8  
    string, 8–12

## W

WAIT instruction, 21–11, 22–2  
Warning level, 4–16  
WIDTH operator, 9–13

## X

.XALL directive, 13–8, 14–7  
XCHG instruction, 17–3  
.XCREF directive, 14–8  
XLAT instruction, 17–3  
.XLIST directive, 14–5  
XOR instruction, 18–17













41640574-000

**CTOS Microsoft Macro Assembler 5.1.1**

product acronyms:

**MASM**

product aliases:

**MASM, Microsoft Assembler**

To order additional copies of this announcement, see Section 7, Ordering Procedure.

---

**Distribution lists:**

**UML: PR5, SA, SN, SU, SW, BT247, BT248**

**URC: 224, 228, 231, 502, 114**

**System: CTOS**

**Release: 5.1.1**

**Part number: 4164 0582-200**

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

BTOS is a trademark; MAPPER, OFIS, and Unisys are registered trademarks; and CUSTOMCARE is a registered service mark of Unisys Corporation.

CTOS is a registered trademark and Shared Resource Processor (SRP) is a trademark of Convergent Technologies Inc., a wholly-owned subsidiary of Unisys Corporation.

MS-DOS and XENIX are registered trademarks of Microsoft Corporation.

OS/2 is a registered trademark of International Business Machines Corporation.

Correspondence regarding this publication should be forwarded to Unisys Corporation by addressing remarks to Product Information, 747 Calle Plano, Camarillo, CA 93012 U.S.A.

# Contents

|                  |                                                    |           |
|------------------|----------------------------------------------------|-----------|
| <b>Section 1</b> | <b>General Description</b>                         | <b>1</b>  |
|                  | <b>Introduction</b>                                | <b>1</b>  |
|                  | <b>Product Description</b>                         | <b>1</b>  |
|                  | BTOS Family of Products                            | 2         |
|                  | CTOS Microsoft Macro Assembler                     | 2         |
| <b>Section 2</b> | <b>Release Functionality</b>                       | <b>3</b>  |
| <b>Section 3</b> | <b>Product Interdependencies</b>                   | <b>5</b>  |
|                  | <b>Interdependent Software</b>                     | <b>5</b>  |
|                  | <b>Interdependent Hardware</b>                     | <b>7</b>  |
|                  | Workstations and Processors                        | 7         |
|                  | Monitors and Graphics                              | 7         |
|                  | Keyboards and Pointing Devices                     | 7         |
|                  | Hard Disk and Diskette Drives                      | 7         |
|                  | <b>Media Sizing</b>                                | <b>8</b>  |
|                  | <b>Random-Access Memory (RAM) Sizing</b>           | <b>8</b>  |
| <b>Section 4</b> | <b>Customer Product Information</b>                | <b>9</b>  |
|                  | CTOS Microsoft Macro Assembler Product Information | 9         |
| <b>Section 5</b> | <b>Migration Requirements</b>                      | <b>13</b> |
|                  | <b>Migration from Previous Levels</b>              | <b>13</b> |
|                  | <b>Operation Environment Compatibility</b>         | <b>13</b> |
|                  | <b>XE530 Compatibility</b>                         | <b>13</b> |
|                  | <b>Upgrade Assistance</b>                          | <b>13</b> |
| <b>Section 6</b> | <b>Restrictions and Known Limitations</b>          | <b>15</b> |
|                  | <b>Existing Items</b>                              | <b>16</b> |
|                  | Features not supported                             | 16        |

|                  |                                                            |           |
|------------------|------------------------------------------------------------|-----------|
| <b>Section 7</b> | <b>Ordering Procedure .....</b>                            | <b>17</b> |
|                  | <b>Hardware and Software Product Orders .....</b>          | <b>17</b> |
|                  | Eligible Unisys SURETY Customers .....                     | 17        |
|                  | Existing Unisys Customers .....                            | 18        |
|                  | Other Unisys Customers .....                               | 19        |
|                  | <b>Product Information Orders .....</b>                    | <b>19</b> |
|                  | Ordering by Part Number .....                              | 19        |
|                  | Ordering a Universal Mailing List (UML) Subscription ..... | 19        |
|                  | <b>Unisys SURETY, a CUSTOMCARE Service .....</b>           | <b>20</b> |
|                  | <b>Unisys Personnel Only .....</b>                         | <b>21</b> |

# Tables

|     |                                                                                                  |    |
|-----|--------------------------------------------------------------------------------------------------|----|
| 3-1 | Interdependent Software .....                                                                    | 6  |
| 3-2 | Media Sizing Considerations .....                                                                | 8  |
| 3-3 | RAM Considerations and Memory Modes .....                                                        | 8  |
| 4-1 | Part Numbers and UML Codes for<br>CTOS Microsoft Macro Assembler 5.1.1 Product Information ..... | 11 |
| 7-1 | Literature Coordinator Contacts .....                                                            | 21 |





# Section 1

## General Description

This section contains:

- An introduction to the Software Release Announcement (SRA)
- A concise description of CTOS Microsoft Macro Assembler 5.1.1 software, including major options and important new or existing features, and information on related families of products

See Section 4, Customer Product Information, and Section 7, Ordering Procedure, for additional ordering information.

## Introduction

Unisys uses a Software Release Announcement (SRA) to announce a new software product release. The SRA provides you with release-specific technical details and ordering information. It complements marketing, sales, and other product information.

Unisys automatically sends the SRA to eligible Unisys SURETY customers and includes it in each software product package. You can order extra copies of the SRA. You can also subscribe to certain Universal Mailing List (UML) codes to receive future Software Release Announcements automatically, as they become available.

The SRA provides you with interdependency and migration considerations and additional sources of information. Use this information for planning decisions and ordering purposes. You can also use this information when you configure and install a software release.

If you are preparing to use a software release, you will find information on available support services. The SRA provides information about known restrictions, limitations, and conditions. To maintain your product information, the SRA provides subscription information.

## Product Description

CTOS Microsoft Macro Assembler 5.1.1 is part of the CTOS (BTOS) family of integrated and cooperating products and services.

## **BTOS Family of Products**

The CTOS family of products includes:

- The modular family of CTOS workstations and components, with built-in cluster networking
- The CTOS (BTOS and BTOS II) multitasking, multiuser operating system, utilities, and system services
- CTOS communications, environmental, and application software
- Unisys customer services
- Line-of-business and general solutions

## **CTOS Microsoft Macro Assembler**

CTOS Microsoft Macro Assembler 5.1.1 is a port of the Microsoft Macro Assembler, version 5.1. It provides all the tools a programmer needs to create assembly-language programs. It provides a logical syntax suited to the segmented architecture of the 8086-family of microprocessors, and the 8087-family of math coprocessors.

The assembler produces relocatable object modules for assembly language source files. These object modules can be linked to create executable programs. Object modules created by the assembler are compatible with other high-level-language object modules including those created with the following compilers:

BTOS Pascal  
BTOS C  
BTOS Fortran  
BTOS Basic  
CTOS Cobol/2  
CTOS High C  
CTOS Microsoft C

## Section 2

# Release Functionality

This is the initial release of CTOS Microsoft Macro Assembler. The *CTOS Microsoft Macro Assembler Programming and Installation Reference Manual* provides complete details about the product features. See the following sections of this SRA for related information:

- Section 3, Product Interdependencies
- Section 6, Restrictions and Limitations



## Section 3

# Product Interdependencies

This section contains:

- Information on interdependent software releases and hardware you need to use CTOS Microsoft Macro Assembler 5.1.1
- Random-access memory (RAM) and media sizing considerations

You can use the details in this section to configure your system. A typical CTOS Microsoft Macro Assembler system will include your choice of:

- A protected mode workstation with at least 400K bytes of RAM
- CTOS hardware modules with a at least 20 megabytes of hard disk and a diskette drive
- CTOS monitor and a keyboard
- CTOS II, BTOS II, or XE Operating System software
- BTOS, BTOS II, or CTOS installable system service, utility, or application software

To order Unisys products or for ordering help, see Section 7, Ordering Procedure.

## Interdependent Software

Table 3-1 lists interdependent software releases you need to use with CTOS Microsoft Macro Assembler 5.1.1. Table 3-1 also describes the nature of the interdependencies and lists style names and release levels (for ordering purposes). For products that have had recent releases, Table 3-1 may also include the previous release level.

**Note:** For CTOS Microsoft Macro Assembler 5.1.1, Unisys fully supports and recommends the releases listed in Table 3-1. If you are using older releases, consider migrating to these releases at the earliest opportunity. Older releases of interdependent software may work with CTOS Microsoft Macro Assembler 5.1.1, but Unisys may not fully support them. If you require a correction to an older release, Unisys may ask you to upgrade to a newer release.

As newer releases become available, Unisys may no longer offer certain ordering and support services for older releases. Review the applicable Software Release Announcement (SRA) for future compatibility and service information. See Section 4, Customer Product Information, and Section 9, Ordering Procedure, for details on how you can order an SRA. These sections also provide subscription information, so you can automatically receive future announcements.

**Table 3-1. Interdependent Software**

| Description                                                                       | Order By Style | Release               |
|-----------------------------------------------------------------------------------|----------------|-----------------------|
| For use on XE Shared Resource Processors (SRP), requires:                         |                |                       |
| CTOS/XE (includes BTOS II 3.2.0)                                                  | XE530-MOS      | 3.0.0 (June 1990)     |
| For use on CTOS workstations, depending on your choice of processor, requires:    |                |                       |
| BTOS II B28 Server Operating System                                               | B28-MOS        | 3.2.0 (June 1990)     |
| BTOS II B38 Server Operating System                                               | B38-MOS        | 3.2.0 (June 1990)     |
| BTOS II B39 Server Operating System                                               | B39-MOS        | 3.2.0 (June 1990)     |
| BTOS II B28 Server Operating System                                               | B28-MOS        | 3.0.2 (August 1989)   |
| BTOS II B38 Server Operating System                                               | B38-MOS        | 3.0.2 (August 1989)   |
| BTOS II B39 Server Operating System                                               | B39-MOS        | 3.0.2 (August 1989)   |
| CTOS II B25 Protected Mode Operating System                                       | B25-POS        | 3.3.0 (May 1991)      |
| CTOS II B25 Development Utilities                                                 | B25-DUX        | 3.3.0 (May 1991)      |
| For multi-context and windowing services, requires one of the following releases: |                |                       |
| BTOS Context/Window Manager                                                       | B25-CM6        | 4.0.0 (July 1990)     |
|                                                                                   |                | 1.3.5 (May 1989)      |
| For networking, requires one of the following releases:                           |                |                       |
| CTOS BNet II                                                                      | B25-BNT        | 2.0.3 (January 1991)  |
| BTOS BNET                                                                         | B25-BN3        | 3.1.1 (February 1988) |
| For BTOS BNET 3.1.1, requires one of the following releases:                      |                |                       |
| BTOS BNA (Burroughs Network Architecture)                                         | B25-NCS        | 1.1.1 (July 1989)     |
| BTOS Local Area Network (BLAN)                                                    | B25-LG3        | 3.0.1 (December 1987) |

# **Interdependent Hardware**

This section provides information on hardware products you need to use CTOS Microsoft Macro Assembler 5.1.1. Hardware includes workstations, monitors, and keyboards. This section also provides random-access memory and media sizing considerations.

## **Workstations and Processors**

CTOS Microsoft Macro Assembler 5.1.1 operates on all protected mode 286, 386 and 486 based processors, protected mode workstations, the XE530, and the SG5000.

## **Monitors and Graphics**

CTOS Microsoft Macro Assembler 5.1.1 requires a monitor. CTOS Microsoft Macro Assembler 5.1.1 runs with all supported video graphics and monitors. CTOS Microsoft Macro Assembler 5.1.1 does not require a graphics module to operate.

## **Keyboards and Pointing Devices**

CTOS Microsoft Macro Assembler 5.1.1 requires a keyboard. You can use K1, K2, K3, K4, and K5 keyboards. CTOS Microsoft Macro Assembler 5.1.1 does not require a mouse to operate.

## **Hard Disk and Diskette Drives**

CTOS Microsoft Macro Assembler 5.1.1 requires a hard disk drive. Table 3-2 lists media sizing considerations. CTOS Microsoft Macro Assembler 5.1.1 requires a 5-1/4-inch floppy diskette drive or a 3-1/2-inch diskette drive for software installation only. However, these drives are useful for storing BTOS files on removable media.



## Media Sizing

Table 3-2 lists approximate media sizing considerations for all required and optional installation files. All media sizes are the largest sizes. Even though you may use less disk space, allow room for work and expansion.

Table 3-2. **Media Sizing Considerations**

| <b>Description</b>                   | <b>Sectors</b> | <b>Bytes</b> |
|--------------------------------------|----------------|--------------|
| CTOS Microsoft Macro Assembler 5.1.1 | 1475           | 755200       |

## Random-Access Memory (RAM) Sizing

As a guideline, Table 3-3 shows the RAM considerations you can use as a typical memory requirement. Your particular use will determine your exact RAM needs.

Using Table 3-3, you can also identify the mode of memory each program uses. Each program uses all available memory in its memory partition.

Table 3-3. **RAM Considerations and Memory Modes**

| <b>Description</b>                   | <b>Memory Mode</b> | <b>KBytes RAM</b> |
|--------------------------------------|--------------------|-------------------|
| CTOS Microsoft Macro Assembler 5.1.1 | Protected          | 400               |

## Section 4

# Customer Product Information

This section contains:

- The names and part numbers of product information you can use with CTOS Microsoft Macro Assembler 5.1.1
- Universal Mailing List (UML) code information for product announcement subscriptions

Unisys packages one set of product information inside each corresponding product package. You can also order product information separately. To order priced copies, or for UML price information and a UML subscription, see Section 7, Ordering Procedure.

## CTOS Microsoft Macro Assembler Product Information

You can order the CTOS Microsoft Macro Assembler software and its associated product information by using style number B25-MSA.

Table 4-1 lists CTOS Microsoft Macro Assembler 5.1.1 product information and the release level to which it relates. For ordering purposes, it lists the product part numbers and for subscription orders, it lists the Universal Mailing List (UML) codes.

If you subscribe to a UML code, you will automatically receive future announcements as they become available. Any announcements you receive relate only to the UML code you choose.

Each UML code represents the following:

- **Announcement Category**

Unisys announces product offerings using a Software Release Announcement (SRA) and a Product Information Announcement (PIA). For a description of the SRA, see Section 1, General Description. The PIA provides a concise description of the literature it relates to. The PIA also provides information on part numbers and UML codes.

- **Release Level**

You can subscribe to UML codes for announcements relative to all future releases.

For each UML code subscription, you can tell Unisys the quantity you want sent to each address. For each UML mailing, Unisys checks for duplicate or overlapping subscription to be sure you receive the correct quantity.

You can prepare for your UML subscription order as follows:

- 1 Review Table 4-1 to check your existing CTOS Microsoft Macro Assembler 5.1.1 product information. Identify how many of each guide or manual you need. List the name and full mailing address of each person who requires a subscription. Make a note of the quantity they require.
- 2 Order any part numbers you need to bring everyone up to date.
- 3 Select the UML code matching your needs and place your subscription order.

Certain Unisys SURETY customers may be eligible to receive, automatically, items for which a UML subscription is also available. Review Section 9, Ordering Procedure, for additional information.

Unisys includes all applicable errata and updates with the latest release of any part number using the same prefix. Updates contain any preceding errata.

**Table 4-1. Part Numbers and UML Codes for  
CTOS Microsoft Macro Assembler 5.1.1 Product Information**

| <b>Description</b>                                                                  | <b>Release</b> | <b>Part Number</b> | <b>UML Codes</b> |                          |
|-------------------------------------------------------------------------------------|----------------|--------------------|------------------|--------------------------|
|                                                                                     |                |                    | <b>SRA Only</b>  | <b>All Announcements</b> |
| Discontinuance Announcements                                                        | ongoing        | -                  | -                | BT247                    |
| Product Information Announcements                                                   | ongoing        | -                  | -                | BT247                    |
| Software Release Announcement                                                       | 5.1.1          | 4164 0582-200      | BT248            | BT247                    |
| Cover Letter (5-1/4-inch media)                                                     | 5.1.1          | 4164 0582-300      | -                | -                        |
| Cover Letter (3-1/2-inch media)                                                     | 5.1.1          | 4164 0582-500      | -                | -                        |
| <i>CTOS Microsoft Macro Assembler Programming and Installation Reference Manual</i> | 5.1.1          | 4164 0574-000      | -                | -                        |

**Note:** The UML codes represent the following:

- BT247 is for subscription to all CTOS Microsoft Macro Assembler announcements (SRA, PIA, and Discontinuance Announcement), for all future releases.
- BT248 is for subscription to all CTOS Microsoft Macro Assembler SRAs only, for all future releases.



## **Section 5**

# **Migration Requirements**

This section provides an overview of hardware and software pre- and post-migration requirements and restrictions. It states whether special upgrade help or procedures are available, required, or recommended.

## **Migration from Previous Levels**

This is the initial release of the CTOS Microsoft Macro Assembler. There are no migration requirements from previous levels.

## **Operating Environment Compatibility**

CTOS Microsoft Macro Assembler 5.1.1 executes only in protected mode operating environments.

## **XE530 Compatibility**

CTOS Microsoft Macro Assembler 5.1.1 executes directly on XE530 processor boards. You can store associated files on the XE530 storage devices.

## **Upgrade Assistance**

For details on available upgrade services and Unisys SURETY eligibility, contact your Unisys representative or reseller.



## **Section 6**

# **Restrictions and Known Limitations**

**This section contains descriptions of certain restrictions, limitations, or conditions known as of the date of release.**

- **Restrictions may include such items as unsupported hardware and software products or functional elements within those products**
- **Known limitations may include elements of implied or stated functionality where that capability does not, to some degree, exist**
- **Conditions can include nuances which may or may not require a detour, caution, or some degree of operator awareness**

**Where possible, the text states whether restrictions, limitations, or conditions are permanent. If they are not, it may state a planned release level (or date) that will lift the restriction. Restrictions, limitations, or conditions may be permanent if they are part of the CTOS Microsoft Macro Assembler design. In this case, they are beyond current price, performance, or technical constraints.**

**Unisys provides this information so you can avoid or work around known restrictions, limitations, or conditions you might otherwise find when using CTOS Microsoft Macro Assembler 5.1.1. The Unisys SureNet electronic bulletin board also provides problem correction and avoidance information.**

**See also, Section 4, Customer Product Information, for additional sources of information. See Section 7, Ordering Procedure, for details on how you can order any additional sources of information.**



## Existing Items

None of the functionality, language features, command-line switches, or object output file contents have been altered from that described in the *CTOS Microsoft Macro Assembler Programming and Installation Reference Manual*. Features not relevant to the CTOS environment, but mentioned in this reference manual are not supported.

## Features not supported

- The PATH environment variable
- References to MS-DOS, OS/2 or XENIX operating systems
- References to MS-DOS, OS/2 or XENIX O/S utilities (e.g. LIB, LINK, EXE2BIN)
- The 'Tiny' segment model for MS-DOS .COM files
- The DOSSEG directive
- The INCLUDELIB directive
- O/S calls via software interrupts (e.g. 21h)
- The /B command line option which sets the buffer size for reading the CTOS MASM source.

# Section 7

## Ordering Procedure

**This section contains:**

- Ordering procedures for new and existing customers
- Information on how to order CTOS Microsoft Macro Assembler 5.1.1 and related software and hardware
- Information on how to order product information, including subscriptions
- Information on ordering procedures for Unisys SURETY services and customers

(Unisys personnel: special product information and UML subscription ordering instructions are at the end of this section.)

## Hardware and Software Product Orders

So each type of customer may use a more efficient method, there are different procedures for ordering software and hardware products. The ordering procedure to use depends on whether you are:

- An eligible Unisys SURETY customer
- An existing customer without a service contract
- A new customer

## Eligible Unisys SURETY Customers

Based on the Unisys Reporting Code (URC) associated with any existing system configuration, Unisys determines which Unisys SURETY customers who are eligible to receive Update Maintenance support for their existing software. For each eligible customer, Unisys prints a customized Update Service Request (USR), and mails it with the Software Release Announcement.

### **The USR**

- Includes a list of the style names available to the Unisys SURETY customer
- Explains how the Unisys SURETY customer can place an order
- Provides a toll-free telephone number for the Unisys SURETY customer to call

If Unisys has attached a USR to this announcement, refer to it for detailed ordering instructions.

### **Existing Unisys Customers**

You can purchase CTOS Microsoft Macro Assembler 5.1.1 directly from the **BTOS Connection** at (800) 344-9038. You must meet the following criteria:

- Not an eligible Unisys SURETY customer (see above for eligibility), or prefer to take advantage of the speedy delivery service offered by the **BTOS Connection**
- Have an account with Unisys
- Do not normally purchase through a Unisys reseller
- Can FAX or mail a purchase order to:

FAX: (408) 434-2154

Mail: Unisys Corporation  
The BTOS Connection  
P.O. Box 6685, Mail Stop 18-009  
San Jose, CA 95150-6685

The **BTOS Connection** can normally ship orders the same day if:

- The products are in the **BTOS Connection** catalog
- The purchase order arrives by 10:00 AM Pacific Time

Call the **BTOS Connection** for a copy of the latest catalog.

## **Other Unisys Customers**

If you do not meet the above criteria, contact your local Unisys representative or reseller. Unisys has branches and representatives in most major cities, worldwide. If you require further assistance, contact:

Unisys Corporation  
Blue Bell, Pennsylvania 19424-0001  
U.S.A.  
(215) 986-4011

## **Product Information Orders**

There are two types of product information related orders. You can order by part number, or you can order a Universal Mailing List (UML) subscription by UML code.

(Unisys personnel: special product information and UML subscription ordering instructions are at the end of this section.)

### **Ordering by Part Number**

To purchase priced copies of the product information listed in this SRA, contact either:

- Unisys Direct at (800) 228-9224
- Your Unisys representative or reseller

### **Ordering a Universal Mailing List (UML) Subscription**

For UML ordering information and to order a UML subscription, contact your Unisys representative or reseller.

## **Unisys SURETY, a CUSTOMCARE Service**

You can purchase support for CTOS Microsoft Macro Assembler product as part of a Unisys SURETY Intro, Basic, BasicPlus, or Comprehensive level of coverage. Relative to this product, Unisys SURETY levels may include the following services:

- Centralized telephone support via a 1-900 or 1-800 number
- Software maintenance releases at a media charge
- Preferred rates on supplies and on-call service
- Access to the Unisys SureNet Public Bulletin Board for electronic trouble shooting

In addition to these services, you can cover many other CUSTOMCARE services under one umbrella contract. Charges vary according to the services you select.

Contact your Unisys representative or reseller for Unisys SURETY and CUSTOMCARE details and price information.

# Unisys Personnel Only

For UML subscriptions or Electronic Literature Ordering (ELO), contact your Literature Coordinator.

You can identify your UML Literature Coordinator by accessing the corporate MAPPER<sup>®</sup> system, DISMAP, and the UML-COORD run. Enter your organization number or cost center and the Payroll Location Code for the facility printing your payroll checks. If you need a UML Literature Coordinator for your location, assign this responsibility to an individual with DISMAP access. Have that person contact UML Maintenance Information Distribution (MID) (see Table 7-1) to request security clearance and instructions.

If you need an ELO Literature Coordinator for your location, assign this responsibility to an individual with DISMAP access. Have that person contact the ELO Customer Service Department as listed in Table 7-1.

Table 7-1. Literature Coordinator Contacts

| <b>Mail</b>                                                                                                                                                    | <b>OFIS Link</b> | <b>Phone</b>                                | <b>FAX</b>                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------------------------------------------|---------------------------------------------|
| <b>Electronic Literature Ordering (ELO)</b>                                                                                                                    |                  |                                             |                                             |
| Unisys Corporation<br>Corporate Software and<br>Publications Operations<br>Customer Service Department<br>13250 Haggerty Road North<br>Plymouth, MI 48170-1892 | CSPO/Corp        | NET <sup>2</sup> 262-4680<br>(313) 451-4680 | NET <sup>2</sup> 2624901<br>(313) 451-4901  |
| <b>Universal Mailing List (UML)</b>                                                                                                                            |                  |                                             |                                             |
| Unisys Corporation<br>Maintenance Information Distribution<br>P.O. Box 500<br>Blue Bell, PA 19424-0035                                                         | MLW2/Corp        | NET <sup>2</sup> 423-6269<br>(215) 986-6269 | NET <sup>2</sup> 423-6892<br>(215) 986-6892 |

10/10/10

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18